Model Recognition as Planning

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

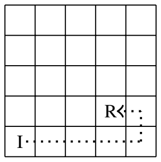

Figure 1: Execution of a robot navigation plan in a 5×5 grid.

```
(:action inc-x
  :parameters (?x1 ?p2)
  :precondition (and (x-coord ?x1) (next ?x1 ?x2) (even))
  :effect (and (not (x-coord ?x1)) (x-coord ?x2)))

(:action dec-x
  :parameters (?x1 ?px2)
  :precondition (and (x-coord ?x1) (next ?x2 ?x1) (odd))
  :effect (and (not (x-coord ?x1)) (x-coord ?x2)))

(:action inc-y-min-odd
  :parameters (?y1 ?y2)
  :precondition (and (y-coord ?y1) (not (next ?y2 ?y1)) (odd))
  :effect (and (not (y-coord ?y1)) (y-coord ?y2) (not (odd)) (even)))

(:action inc-y-min-even
  :parameters (?y1 ?y2)
  :precondition (and (y-coord ?y1) (not (next ?y2 ?y1)) (even))
  :effect (and (not (y-coord ?y1)) (y-coord ?y2) (not (even)) (odd)))

(:action inc-y-max-odd
  :parameters (?py1 ?py2)
  :precondition (and (y-coord ?y1) (not (next ?y1 ?y2)) (odd))
  :effect (and (not (y-coord ?y1)) (y-coord ?y2) (not (odd)) (even)))

(:action inc-y-max-even
  :parameters (?y1 ?y2)
  :precondition (and (y-coord ?y1) (not (next ?y1 ?y2)) (even))
  :effect (and (not (y-coord ?y1)) (y-coord ?y2) (not (even)) (odd)))
```

Figure 2: Action model for a robot navigation in a $n \times n$ grid.

Abstract

Given a partially observed plan execution and a set of possible planning models (share same state variables but define different action models), *model recognition* is the task of identifying which model in the set produced the observed plan execution. The paper formalizes the *model recognition* task and proposes a novel method to assess the probability of a STRIPS model to produce a given partially observed plan execution. This method, that we called *model recognition as planning*, is robust to missing data in the given observed plan execution (i.e. missing intermediate states and actions) besides, it is computable with an off-the-shelf classical planner. The effectiveness of *model recognition as planning* is shown in a set of STRIPS models encoding different *regular automatae* and different *Turing machines*. We show that *model recognition as planning* succeeds to identify the executed automata despite the internal machine states or actual applied transitions, are unknown.

Introduction

Plan recognition is the task of predicting the future actions of an agent provided observations of its current behaviour. Plan recognition is considered *automated planning* in reverse; while automated planning aims to compute a sequence of actions that accounts for a given goals, plan recognition aims to compute the goals that account for an observed sequence of actions (Geffner and Bonet 2013).

Diverse approaches has been proposed for plan recognition such as *rule-based systems*, *parsing*, *graph-covering*, *Bayesian nets*, etc (Carberry 2001). *Plan recognition as planning* is the model-based approach for plan recognition (Ramírez 2012; Ramírez and Geffner 2009). This approach assumes that the action model of the observed agent is known and leverages it to compute the most likely goal of the agent, according to the observed plan execution.

This paper introduces the task of *model recognition*. Given a partially observed plan execution and a set of possible planning models (share the same state variables but define different action models), *model recognition* is the task of identifying which model in the set has the highest probability of producing the input observation. To better illustrate *model recognition*, imagine a robot in a $n \times n$ grid whose navigation is determined by the STRIPS model shown in Figure 2. According to this model the robot increments its *x-coordinate* when it is at an *even* row while, at *odd rows*, decrements the *x-coordinate*. Further, the robot exclusively increments the *y-coordinate* either at the *end* or at the *begining* of a row. Apart from this particular navigation model, different models can be defined within the same state variables and these models can determine different kinds of robot navigation. Given an observation of a plan execution (e.g, the six-step plan illustrated by Figure 1) the *model recognition* task aims here to identify which navigation model produced that observation.

Model recognition is of interest because once the planning model is recognized, then the model-based machinery for automated planning becomes applicable (Ghallab, Nau, and Traverso 2004). In addition, it enables indentifying different kinds of automatae by observing their execution (it is well-known that diverse automatae representations, like *finite state controllers*, *push-down automata* or GOLOG *programs*, can be encoded as classical planning models (Baier, Fritz, and McIlraith 2007; Bonet, Palacios, and Geffner 2010; Segovia-Aguas, Jiménez, and Jonsson 2017)).

The paper introduces also *model recognition as planning*; a novel method to assess the probability of a given STRIPS model to produce an observed plan execution. The method is robust to missing data in the intermediate states and actions of the observed plan execution besides, it is computable with an off-the-shelf classical planner. The paper evaluates the effectiveness of *model recognition as planning* with a set of STRIPS models that represent different *regular automatae* and different *Turing Machines*. All of these *automatae* are defined within the same *alphabet* and same *machine states* but different *transition functions*. We show that *model recognition as planning* succeeds to identify the executed *automata* despite its internal machine state or actual applied transitions are unknown.

Background

This section formalizes classical planning and the observation of the execution of a classical plan.

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 (without loss of generality, we will 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 full assignment of values to fluents and we explicitly include negative literals $\neg f$ in states; i.e. $|s| = |F|$, so the size of the state space is $2^{|F|}$. Like in PDDL (Fox and Long 2003), we assume that fluents F are instantiated from a set of *predicates* Ψ. 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 induced by assigning objects in Ω to the arguments of predicates in Ψ; i.e. $F = \{p(\omega) : p \in \Psi, \omega \in \Omega^{ar(p)}\}$ such that Ω^k is the k-th Cartesian power of Ω.

A *classical planning frame* is a pair $\langle F, A\rangle$, where F is a set of fluents and A is a set of actions whose dynamics are specified with two functions: $App(s)$ that denotes the subset of actions applicable in a state s and $\theta(s,a)$ that denotes the *successor state* that results of applying a in s. In this work we specify action dynamics with the STRIPS model. With this regard, an action $a \in A$ is defined with:

- $\mathsf{pre}(a) \in \mathcal{L}(F)$, the *preconditions* of a, is the set of literals that must hold for the action $a \in A$ to be applicable.
- $\mathsf{eff}^+(a) \in \mathcal{L}(F)$, the *positive effects* of a, is the set of literals that are true after the application of action $a \in A$.
- $\mathsf{eff}^-(a) \in \mathcal{L}(F)$, the *negative effects* of a, is the set of literals that are false after the application of the action.

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

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 the *state trajectory* $s = \langle s_0, s_1, \ldots, s_n\rangle$ 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 length* is denoted with $|\pi| = n$. A plan π *solves* P iff $G \subseteq s_n$, i.e., if the goal condition is satisfied at the last state reached after following the application of the plan π in the initial state I. A solution plan for P is *optimal* if it has minimum length.

Conditional effects

An action $a \in A$ with conditional effects is defined as a set of *preconditions* $\mathsf{pre}(a) \in \mathcal{L}(F)$ 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 a state s iff $\mathsf{pre}(a) \subseteq s$, and the *triggered effects* resulting from the action application are the effects whose conditions hold in s:

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

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

The observation model

Given a classical planning problem $P = \langle F, A, I, G\rangle$ and a plan π that solves P, the observation of the execution of π on P is $\tau = \langle s_1, a_1, \ldots, a_n, s_m\rangle$, an interleaved combination of $1 \leq m \leq |\pi| + 1$ observed states and $1 \leq n \leq |\pi|$ observed actions such that:

- Observed states may be partial states. The value of certain fluents may be omitted in that states, i.e. $|s_i| \leq |F|$ for every $0 \leq i \leq m$.
- The sequence of observed states $\langle s_1, \ldots, s_m\rangle$ in τ is the same sequence of states traversed by π but certain states may also be omitted. Therefore the transitions between two consecutive observed states in τ may require the execution of more than a single action. Formally, $\theta(s_i, \langle a_1, \ldots, a_k\rangle) = s_{i+1}$, where $k \geq 1$ is unknown and unbound. This means that having τ does not implies knowing the actual length of π.
- The sequence of observed actions $\langle a_1, \ldots, a_n\rangle$ in τ is a sub-sequence of the solution plan π.

Definition 1 (Φ-observation)**.** *Given a subset of fluents $\Phi \subseteq F$ we say that τ is a Φ-observation of the execution of π on P iff, for every $0 \leq i \leq m$, each observed state s_i only contains fluents in Φ.*

Model Recognition

The *model recognition* task is a tuple $\langle P, M, \tau\rangle$ where:

- $P = \langle F, A, I, G\rangle$ is a classical planning problem such that the dynamics of the actions in A is unknown because functions $App(s)$ and/or $\theta(s, a)$ are undefined.
- $M = \{\mathcal{M}_1, \dots, \mathcal{M}_m\}$ is a finite and non-empty set of models for the actions in A. Each model in $\mathcal{M} \in M$ defines a different function pair (*App*,θ).
- τ is an observation of the execution of a solution plan π for P.

A solution to the *model recognition* task is the discrete probability distribution $P(\mathcal{M}|\tau)$ that expresses, for each model $\mathcal{M} \in M$, its probability of producing the observation τ.

According to the *Bayes* rule, the probability of an hypothesis $\mathcal{H}$, provided the observation $\mathcal{O}$, is given by the expression $P(\mathcal{H}|\mathcal{O}) = \frac{P(\mathcal{O}|\mathcal{H})P(\mathcal{H})}{P(\mathcal{O})}$. In the *model recognition* task, hypotheses are about the possible action models $\mathcal{M} \in M$ while the given observation is the partially observed plan execution τ. The $P(\mathcal{M}|\tau)$ probability distribution can then be estimated in three steps:

1. Computing the *a priori* probabilities. $P(\tau)$, that measures how surprising is the given observation and $P(\mathcal{M})$, that expresses if one model is a priori more likely than the others.
2. Computing the conditional probability $P(\tau|\mathcal{M})$. Our approach is to estimate this value according to the minimum *amount of edits* that is required by the model $\mathcal{M}$ to produce a plan π_τ such that τ is an observation of the execution of π_τ on the classical planning problem P. The *edit distance* is a similarity metric, traditionally computed over strings or graphs, that has been proved succesful for *pattern recognition* (Masek and Paterson 1980; Bunke 1997). In this work this similarity metric refers to the edition of classical planning models.
3. Applying the Bayes rule to obtain the normalized posterior probabilities (the $P(\mathcal{M}|\tau)$ probabilities must sum 1 for all the $\mathcal{M} \in M$).

Recognition of STRIPS models

Here we analyze the particular instantiation of the *model recognition* task where the dynamics of the actions A (i.e. the *App* and θ functions) are specified with STRIPS action schemas like the ones shown in Figure 2.

The space of STRIPS models

We assume that $\mathsf{eff}^-(a) \subseteq \mathsf{pre}(a)$, $\mathsf{eff}^-(a) \cap \mathsf{eff}^+(a) = \emptyset$ and $\mathsf{pre}(a) \cap \mathsf{eff}^+(a) = \emptyset$ and that actions $a \in A$ are instantiated from given action schemas, as in PDDL.

Like in PDDL (McDermott et al. 1998; Fox and Long 2003), we assume that fluents F are instantiated from a set of *predicates* Ψ. 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 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.

```
(pre_x-pos_inc-x_v1) (pre_next_inc-x_v1_v2) (pre_even_inc-x_)
(del_x-pos_inc-x_v1) (add_x-pos_inc-x_v2)
```

Figure 3: Propositional encoding for the `inc-x` schema previously defined in Figure 2.

Let $\Omega_v = \{v_i\}_{i=1}^{\max_{a\in A} ar(a)}$ be an additional set of objects ($\Omega \cap \Omega_v = \emptyset$), that we denote as *variable names*, and that is bound by the maximum arity of an action in a given planning frame. For instance, in a 5×5 grid $\Omega = \{p_1, p_2, p_3, p_4, p_5\}$ while $\Omega_v = \{v_1, v_2\}$ because all the actions have arity two. We define F_v, a new set of fluents, $F \cap F_v = \emptyset$, produced instantiating Ψ using only *variable names*. This set contains ten elements for the mentioned navigation model, F_v=\{`x-pos(`v_1`)`, `x-pos(`v_2`)`, `y-pos(`v_1`)`, `y-pos(`v_2`)`, `odd()`, `even()`, `next(`v_1`,`v_1`)`, `next(`v_1`,`v_2`)`, `next(`v_2`,`v_1`)`, `next(`v_2`,`v_2`)`\}.

For a given action schema ξ, then $F_\xi \subseteq F_v$ defines the subset of elements that can appear in the preconditions and effects of that action schema. For instance $F_{\texttt{inc-x}} = F_{\texttt{dec-x}} = F_v$ because all actions have arity two.

Now we are ready to define the fluents $pre_f(\xi)$, $del_f(\xi)$ and $add_f(\xi)$, for every $f \in F_\xi$. These fluents represent a propositional encoding for the preconditions, negative and positive effects of an action schema ξ and hence, they specify the *App* and θ functions for all the actions $a \in A$ built from the corresponding STRIPS action schemas. In more detail, if a fluent $pre_f(\xi)/del_f(\xi)/add_f(\xi)$ holds, it means that f is a precondition/negative/positive effect in that schema. For instance, Figure 3 shows the conjunction of fluents representing the propositional encoding for the preconditions, negative and positive effects of the *inc-x* schema previously defined in Figure 2.

The size of the space of possible STRIPS models for an action schema ξ is given by the expression, $2^{2\times|F_\xi|}$ (STRIPS constraints require negative effects appearing as preconditions, negative effects cannot be positive effects at the same time and also, positive effects cannot appear as preconditions). For the mentioned navigation model, $2^{2\times|F_\xi|} = 1,048,576$ for every action schema. We say that two STRIPS schemes ξ and ξ' are *comparable* iff both share the same space of possible STRIPS models (same parameters and same constraints). For instance, we claim that all the schemas in the mentioned navigation model are *comparable* while `stack` and `pickup` from a four opertator *blocksworld* (Slaney and Thiébaux 2001) are not. Last but not least, two STRIPS models $\mathcal{M}$ and $\mathcal{M}'$ are *comparable* iff there exists a bijective function $\mathcal{M} \mapsto \mathcal{M}^*$ that maps every action schema $\xi \in \mathcal{M}$ to a comparable schema $\xi' \in \mathcal{M}'$ and vice versa.

The STRIPS edit distance

We define two edit *operations* on a STRIPS model $\mathcal{M} \in M$:

- *Deletion*. A fluent $pre_f(\xi)/del_f(\xi)/add_f(\xi)$ is removed from the operator schema $\xi \in \mathcal{M}$, such that $f \in F_\xi$.
- *Insertion*. A fluent $pre_f(\xi)/del_f(\xi)/add_f(\xi)$ is added to the operator schema $\xi \in \mathcal{M}$, s.t. $f \in F_\xi$.

We can now formalize an *edit distance* that quantifies how similar two given STRIPS action models are. The distance is symmetric and meets the *metric axioms* provided that the two *edit operations*, deletion and insertion, have the same positive cost.

Definition 2. *Let $\mathcal{M}$ and $\mathcal{M}'$ be two* comparable STRIPS *action models. The* **edit distance** $\delta(\mathcal{M}, \mathcal{M}')$ *is the minimum number of* edit operations *that is required to transform $\mathcal{M}$ into $\mathcal{M}'$.*

Since F_v is a bound set, the maximum number of edits that can be introduced to a given action model defined within F_v is bound as well.

Definition 3. *The* ***maximum edit distance*** *of an* STRIPS *model $\mathcal{M}$ built from the set of possible elements F_v is $\delta(\mathcal{M}, *) = \sum_{\xi \in \mathcal{M}} 3 \times |F_\xi|$.*

We define now an edit distance to asses the matching of a STRIPS model with respect to an observation of a plan execution.

Definition 4. *Given τ, an observation of the execution of a plan for solving P and $\mathcal{M}$, a* STRIPS *action model built from F_v. The* **observation edit distance**, *$\delta(\mathcal{M}, \tau)$, is the minimal edit distance from $\mathcal{M}$ to any* comparable *model $\mathcal{M}'$ s.t. $\mathcal{M}'$ produces a plan π^*_τ optimal for P and compliant with τ;*

$$\delta(\mathcal{M}, \tau) = \min_{\forall \mathcal{M}' \rightarrow \tau} \delta(\mathcal{M}, \mathcal{M}')$$

The $\delta(\mathcal{M}, \tau)$ distance could also be defined assessing the edition required by the observed plan execution to match the given model. This implies defining *edit operations* that modify τ instead of $\mathcal{M}$ (Sohrabi, Riabov, and Udrea 2016). Our definition of the *observation edit distance* is more practical since normally F_v is smaller than F. In practice, the number of *variable objects* is smaller than the number of objects in a planning problem.

The $P(\mathcal{M}|\tau)$ probability for STRIPS models

Since we assume that the dynamics of the actions $A \in P$ is specified with STRIPS action schemas, we can estimate the $P(\mathcal{M}|\tau)$ probability as follows:

1. Assuming *a priori* all action models $\mathcal{M} \in M$ are equiprobable. There are no reasons to assume that one model is *a priori* more likely than the others so, $P(\mathcal{M}) = \frac{1}{\prod_{\xi \in \mathcal{M}} 2^{2 \times |F_\xi|}}$.
2. Assuming that all the observations of plan executions with maximum n observed actions and m observed states are equiprobable then, $P(\tau) = \frac{1}{|A|^n \times 2^{m \times |F|}}$.
3. Estimating the conditional probability $P(\tau|\mathcal{M})$ by mapping the *observation edit distance* into a $[0, 1]$ likelihood, $1 - \frac{\delta(\mathcal{M}, \tau)}{\delta(\mathcal{M}, *)}$.

```
00 : (insert_add_handempty_stack)
01 : (insert_add_clear_stack_var1)
02 : (apply_unstack blockB blockA i1 i2)
03 : (apply_putdown blockB i2 i3)
04 : (apply_pickup blockA i3 i4)
05 : (apply_stack blockA blockB i4 i5)
06 : (validate_1)
```

Figure 4: Plan for editing (steps [0-1]) and validating (steps [2-6]) a given STRIPS planning model for the *blockswold*.

Model recognition as planning

This section shows that, when the dynamics of the actions $A \in P$ are specified with STRIPS action schemas, then $\delta(\mathcal{M}, \tau)$ can be computed with a compilation of a classical planning with conditional effects. The intuition behind this compilation is that a solution to the resulting classical planning task is a sequence of actions that:

1. **Edits the action model $\mathcal{M}$ to build $\mathcal{M}'$.** A solution plan starts with a *prefix* that modifies the preconditions and effects of the action schemes in $\mathcal{M}$ using to the two *edit operations* defined above, *deletion* and *insertion*.
2. **Validates the edited model $\mathcal{M}'$.** The solution plan continues with a postfix that:
   (a) Induces an optimal solution plan π^*_τ for the original classical planning problem P.
   (b) Validates that τ is an observation of the execution of π^*_τ on the classical planning problem P.

Figure 4 shows the plan with a prefix (steps [0,1]) for editing a given *blockswold* model where the positive effects `(handempty)` and `(clear ?v1)` of the `stack` schema are missing. The postfix of the plan (steps [2,6]) validates the edited action model at the observation of a four action plan for inverting a two-block tower where intermediate states, s_1, s_2 and s_3, are unobserved.

Note that our interest is not in $\mathcal{M}'$, the edited model resulting from the compilation, but in the number of required *edit operations* (insertions and deletions) required by $\mathcal{M}'$ to be validated. In the example of Figure 4 $\delta(\mathcal{M}, \tau) = 2$ and $\delta(\mathcal{M}, *) = 3 \times 2 \times (11 + 5)$ since there are 4 action schemes (`pickup`, `putdown`, `stack` and `unstack`) s.t. $|F_v| = |F_{\texttt{stack}}| = |F_{\texttt{unstack}}| = 11$ while $|F_{\texttt{pickup}}| = |F_{\texttt{putdown}}| = 5$.

The compilation formalization

Conditional effects allow us to compactly define our compilation. Given a STRIPS model $\mathcal{M} \in M$ and the observation τ of the execution of a plan for solving $P = \langle F, A, I, G \rangle$, our compilation outputs a classical planning task with conditionl effects $P' = \langle F', A', I', G' \rangle$ such that:

- F' contains:
  - The original fluents F.
  - Fluents $pre_f(\xi)/del_f(\xi)/add_f(\xi)$ modeling the space of STRIPS models.
  - The fluents $F_\pi = \{plan(name(a_i), \Omega^{ar(a_i)}, i)\}_{1 \leq i \leq n}$ to code the i^{th} action in τ. The static facts $next_{i,i+1}$

and the fluents at_i, $1 \leq i < n$, are also added to iterate through the n steps of τ.

  – The fluents $\{test_j\}_{1 \leq j \leq m}$, indicating the state observation $s_j \in \tau$ where the action model is validated.
  – The fluents $mode_{edit}$ and $mode_{val}$ to indicate whether the operator schemas are edited or validated.

- I' extends the original initial state I with the fluent $mode_{edit}$ set to true as well as the fluents F_π plus fluents at_1 and $\{next_{i,i+1}\}$, $1 \leq i < n$, for tracking the plan step where the action model is validated. Our compilation assumes that initially $\mathcal{M}'$ is defined as $\mathcal{M}$. Therefore fluents $pre_f(\xi)/del_f(\xi)/add_f(\xi)$ hold as given by $\mathcal{M}$.
- $G' = G \bigcup \{at_n, test_m\}$.
- A' comprises three kinds of actions with conditional effects:

1. Actions for *editing* operator schema $\xi \in \mathcal{M}$:
   – Actions for adding a *precondition* $f \in F_v(\xi)$ from the action schema $\xi \in \mathcal{M}$.

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

   – Actions for adding a *negative* or *positive* effect $f \in F_v(\xi)$ to the action schema $\xi \in \mathcal{M}$.

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

   Besides these actions, the A' set also contains the actions for *deleting* a precondition and a negative/positive effect.

2. Actions for *applying* an edited operator schema $\xi \in \mathcal{M}$ bound with objects $\omega \subseteq \Omega^{ar(\xi)}$. Since operators headers are given as input, the variables $pars(\xi)$ are bound to the objects in ω that appear at the same position. Figure 5 shows the PDDL encoding of the action for applying a programmed operator $stack$ from *blocksworld*.

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

   When the observation τ includes observed actions, then the extra conditional effects $\{at_i, plan(name(a_i), \Omega^{ar(a_i)}, i)\} \rhd \{\neg at_i, at_{i+1}\}_{\forall i \in [1,n]}$ are included in the $\mathsf{apply}_{\xi,\omega}$ actions to validate that actions are applied, exclusively, in the same order as they appear in τ.

3. Actions for *validating* the partially observed state $s_j \in \tau$, $1 \leq j < m$.

$$\begin{aligned}\mathsf{pre}(\mathsf{validate_j}) =&s_j \cup \{test_{j-1}\},\\ \mathsf{cond}(\mathsf{validate_j}) =&\{\emptyset\} \rhd \{\neg test_{j-1}, test_j, \neg mode_{val}\}.\end{aligned}$$

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

Figure 5: PDDL action for applying an already programmed schema *stack* (implications are coded as disjunctions).

Evaluation

To evaluate the empirical performance of *model recognition as planning* we defined a set of possible STRIPS models, each representing a different *Turing Machine*, but all sharing the same set of *machine states* and same *tape alphabet*.

Experimental setup

We randomly generated a $M = \{\mathcal{M}_1, \ldots, \mathcal{M}_{100}\}$ set of one-hundred different *Turing Machines* where each $\mathcal{M} \in M$ is a seven-symbol six-state *Turing Machine*. The classical planning frame $\Phi = \langle F, A\rangle$ is the same for all the *Turing Machines*, there is an $a \in A$ action for each pair of tape symbol and non-terminal state machine, while the $\theta(s, a)$ function is defined differently for each the machines (using a different STRIPS action model).

We randomly choose a machine $\mathcal{M} \in M$ and produce the observation τ of a fifty-step execution plan. Finally, we follow our *model recognition as planning* method to identify the *Turing Machine* that produced τ. This experiment is repeated for different amounts of missing information in the input trace τ: unknown applied transitions, unknown internal machine state and unknown values of several tape cells.

Reproducibility MADAGASCAR is the classical planner we used to solve the instances that result from our compilations for its ability to deal with dead-ends (Rintanen 2014). Due to its SAT-based nature, MADAGASCAR can apply the actions for editing preconditions in a single planning step (in parallel) because there is no interaction between them. Actions for editing effects can also be applied in a single planning step, thus significantly reducing the planning horizon.

The compilation source code, evaluation scripts and benchmarks (including the used training and test sets) are fully available at this anonymous repository so any experimental data reported in the paper can be reproduced.

Recognition of *Regular Automatae*

We analyze now *model recognition* when the input set of given set of models represent different *regular automatae*.

A *Regular automatae* is a tuple $\mathcal{M} = \langle Q, q_o, Q_\perp, \Sigma\Box, \delta\rangle$:

- Q is a finite and non-empty set of machine states with the *initial state* $q_0 \in Q$ and the *terminal states* $Q_\perp \subseteq Q$.
- Σ is the *input alphabet*, a finite non-empty set of symbols and the *blank symbol* $\Box \in \Upsilon$ (the only symbol allowed to occur on the tape infinitely often).
- $\delta : \Sigma \times (Q \backslash Q_\perp) \rightarrow \Sigma \times Q \times \{left, right\}$ is the *transition function*. For each pair of tape symbol and non-terminal machine state δ defines: (1), the tape symbol to print at the current position of the header (2), the new state of the machine and (3), whether the header is shifted *left* or *right* after the print operation. If δ is not defined for the current pair of tape symbol and machine state, the machine halts.

Figure 6 illustrate a four-symbol four-state *regular automata* for recognizing the $\{(abc)^n : n \geq 1\}$ language. The *input alphabet* is $\Sigma = \{a, b, c, \Box\}$, and the machine states are $Q = \{q_0, q_1, q_2, \underline{q_3}\}$ (where $\underline{q_3}$ is the only acceptor state).

$$\begin{tikzcd}
Q_0 \arrow[r, "a"] & Q_1 \arrow[r, "b"] & Q_2 \arrow[r, "c"] & Q_3 \arrow[ll, bend left, "a"]
\end{tikzcd}$$

Figure 6: Four-symbol four-state *regular automata* for recognizing the $\{(abc)^n : n \geq 1\}$ language.

```
(:action transition-1        ;;; a,q0 → q1
  :parameters (?x ?xr)
  :precondition (and (head ?x) (symbol-a ?x) (state-q0)
                     (next ?x ?xr))
  :effect (and (not (head ?x))
               (not (symbol-a ?x)) (not (state-q0))
               (head ?xr) (state-q1)))
```

Figure 7: STRIPS action schema that models the transition $q_0 \rightarrow q_1$ of the automata defined in Figure 6.

The STRIPS action schema of Figure **??** models the rule $a, q_0 \rightarrow q_1$ of the *regular automatae* defined in Figure 6. The full encoding of the *automata* of Figure 6 produces a total of sixteen STRIPS action schema.

Recognition of *Turing Machines*

We analyze now *model recognition* when the input set of given set of models represent different *Turing machines*.

A *Turing machine* is a tuple $\mathcal{M} = \langle Q, q_o, Q_\perp, \Sigma, \Upsilon, \Box, \delta\rangle$:

- Q is a finite and non-empty set of machine states with the *initial state* $q_0 \in Q$ and the *terminal states* $Q_\perp \subseteq Q$.
- Σ is the *tape alphabet*, a finite non-empty set of symbols with the *input alphabet* $\Upsilon \subseteq \Sigma$ (symbols allowed to initially appear in the tape) and the *blank symbol* $\Box \in \Upsilon$ (the only symbol allowed to occur on the tape infinitely often).
- $\delta : \Sigma \times (Q \backslash Q_\perp) \rightarrow \Sigma \times Q \times \{left, right\}$ is the *transition function*. For each pair of tape symbol and non-terminal machine state δ defines: (1), the tape symbol to print at the current position of the header (2), the new state of the machine and (3), whether the header is shifted *left* or *right* after the print operation. If δ is not defined for the current pair of tape symbol and machine state, the machine halts.

Figure 8 illustrate a seven-symbol six-state *Turing Machine* for recognizing the $\{a^n b^n c^n : n \geq 1\}$ language. The *tape alphabet* is $\Sigma = \{a, b, c, x, y, z, \Box\}$, the *input alphabet* $\Upsilon = \{a, b, c, \Box\}$ and the machine states are $Q = \{q_0, q_1, q_2, q_3, q_4, \underline{q_5}\}$ (where $\underline{q_5}$ is the only acceptor state).

The STRIPS action schema of Figure 9 models the rule $a, q_0 \rightarrow x, r, q_1$ of the *Turing Machine* defined in Figure 8. The full encoding of the *Turing Machine* of Figure 8 produces a total of sixteen STRIPS action schema.

With regard to our STRIPS model for *Turing Machines*, executions of a *Turing Machine* are definable as an action sequence $\langle a_1, \ldots, a_n\rangle$ that induces the *state trajectory* $\langle s_0, s_1, \ldots, s_n\rangle$ such that a_i ($1 \leq i \leq n$) is applicable in s_{i-1} and generates the successor state $s_i = \theta(s_{i-1}, a_i)$. For instance, the execution of the *Turing Machine* defined

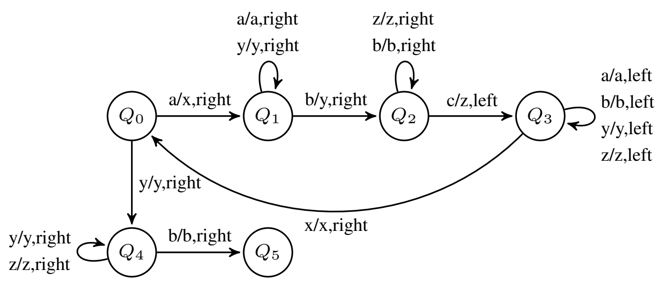

Figure 8: Seven-symbol six-state *Turing Machine* for recognizing the $\{a^nb^nc^n : n \geq 1\}$ language.

```
(:action transition-1        ;;; a,q0 → x,r,q1
  :parameters (?xl ?x ?xr)
  :precondition (and (head ?x) (symbol-a ?x) (state-q0)
                     (next ?xl ?x) (next ?x ?xr))
  :effect (and (not (head ?x))
               (not (symbol-a ?x)) (not (state-q0))
               (head ?xr) (symbol-x ?x) (state-q1)))
```

Figure 9: STRIPS action schema that models the transition $a, q_0 \rightarrow x, r, q_1$ of the Turing Machine defined in Figure 8.

in Figure 8 with an initial tape $abc\square\square\square\dots$ produces the eight-action plan $(a, q_0 \rightarrow x, r, q_1)$, $(b, q_1 \rightarrow y, r, q_2)$, $(c, q_2 \rightarrow z, l, q_3)$, $(y, q_3 \rightarrow y, l, q_3)$, $(x, q_3 \rightarrow x, r, q_0)$, $(y, q_0 \rightarrow y, r, q_4)$, $(z, q_4 \rightarrow z, r, q_4)$, $(\square, q_4 \rightarrow \square, r, \underline{q_5})$.

Assuming that the actual applied transitions is unknown means that the observation τ of the execution of a Turing Machine contains no actions, it is simply a sequence of states $\tau = \langle s_1, \dots, s_m \rangle$. Further, assuming that the internal machine state is unknown means that τ is a Φ-observation and that the Φ subset does not contain `(state-`q`)` fluents, with $q \in Q$ and $q \neq q_0$. Finally, assuming that the values of several tape cells is unknown means that fluents of the kind `(symbol-`σ `?x)` are missing (i.e. unobserved) for some state $s_i \in \tau$ s.t. $1 \leq i \leq m$. These facts affects to the a priori probaility of the possible observations.

On the other hand, the model edition for Turing Machines is limited to these subset of possible positive effects: `(head ?xr)` or `(head ?xl)`, `(symbol-`σ `?x)` with $\sigma \in \Sigma$ and last but not least, `(state-q)` with $q \in Q$. No other positive effects, preconditions, or negative effects are required to be edited. This fact reduces the possible edition operations and affects to the a priori probaility of the possible action models.

Results

Conclusions

The task of *model recognition* can also be understood as a classification task where each class is represented with a different planning model and the observed plan execution is the example to classify. With this regard, planning model that is associated to a class is acting as a class prototype that the summarizes all the plan executions that could be synthesized with that model or in other words, all the examples that belong to that class.

In this work we do not assume that the observed agent is acting rationally, like in *plan recognition as planning* (Ramírez 2012; Ramírez and Geffner 2009). A related approach is recently followed for *model reconciliation* (Chakraborti et al. 2017) where model edition is used to conform the PDDL models of two different agents.

References

Baier, J. A.; Fritz, C.; and McIlraith, S. A. 2007. Exploiting procedural domain control knowledge in state-of-the-art planners. In *ICAPS*, 26–33.

Bonet, B.; Palacios, H.; and Geffner, H. 2010. Automatic derivation of finite-state machines for behavior control. In *AAAI Conference on Artificial Intelligence*.

Bunke, H. 1997. On a relation between graph edit distance and maximum common subgraph. *Pattern Recognition Letters* 18(8):689–694.

Carberry, S. 2001. Techniques for plan recognition. *User Modeling and User-Adapted Interaction* 11(1-2):31–48.

Chakraborti, T.; Sreedharan, S.; Zhang, Y.; and Kambhampati, S. 2017. Plan explanations as model reconciliation: Moving beyond explanation as soliloquy. In *IJCAI*.

Fox, M., and Long, D. 2003. PDDL2.1: An extension to PDDL for expressing temporal planning domains. *Journal of Artificial Intelligence Research* 20:61–124.

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.

Masek, W. J., and Paterson, M. S. 1980. A faster algorithm computing string edit distances. *J. Comput. Syst. Sci.* 20(1):18–31.

McDermott, D.; Ghallab, M.; Howe, A.; Knoblock, C.; Ram, A.; Veloso, M.; Weld, D.; and Wilkins, D. 1998. PDDL – The Planning Domain Definition Language.

Ramírez, M., and Geffner, H. 2009. Plan recognition as planning. In *International Joint conference on Artifical Intelligence*, 1778–1783.

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

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

Segovia-Aguas, J.; Jiménez, S.; and Jonsson, A. 2017. Generating context-free grammars using classical planning. In *International Joint Conference on Artificial Intelligence, ICAPS-17*.

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

Sohrabi, S.; Riabov, A. V.; and Udrea, O. 2016. Plan recognition as planning revisited. In *IJCAI*, 3258–3264.