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

Abstract

Given a set of planning models and a partially observed plan execution, we define *model recognition* as the task of identifying which model in the set has the highest probability of producing the input observation. The paper formalizes the *model recognition* task and proposes a method to assess the probability of a given STRIPS model to produce a partially observed plan execution. This method, that we called *model recognition as planning*, 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 effectiveness of *model recognition as planning* is shown in a set of STRIPS models encoding different *Turing Machines*. We show that *model recognition as planning* succeeds to identify the executed *Turing Machine* despite the actual applied transitions, the internal machine state or the values of several tape cells, 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 according to the observed plan execution.

In this paper we introduce the task of *model recognition*. Given a set of planning models and a partially observed plan execution, *model recognition* is the task of identifying which model in the set has the highest probability of producing the input 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).
- It enables recognizing algorithms by observing their execution. Planning models can encode diverse forms of algorithm representation, like GOLOG *programs*, *finite state controllers* or *push-down automata* (Baier, Fritz, and McIlraith 2007; Bonet, Palacios, and Geffner 2010; Segovia-Aguas, Jiménez, and Jonsson 2017).

This paper introduces *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 *Turing Machines* (all of them defined within the same *tape alphabet* and same *machine states* but different *transition functions*). We show that *model recognition as planning* succeeds to identify the executed *Turing Machine* despite the actual applied transitions, the internal machine state or the values of several tape cells, are unknown.

Background

This section formalizes classical planning and characterizes the space of STRIPS models.

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 contain conflicting values). 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; $|s| = |F|$, so the size of the state space is $2^{|F|}$. Explicitly including negative literals $\neg f$ in states simplifies subsequent definitions but often we will abuse of notation by defining a state s only in terms of the fluents that are true in s, as it 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. An action $a \in A$ is defined with *preconditions*, $\mathsf{pre}(a) \subseteq \mathcal{L}(F)$, *positive effects*, $\mathsf{eff}^+(a) \subseteq \mathcal{L}(F)$, and *negative effects* $\mathsf{eff}^-(a) \subseteq$

```
(:action stack
 :parameters (?v1 ?v2 - object)
 :precondition (and (holding ?v1) (clear ?v2))
 :effect (and (not (holding ?v1)) (not (clear ?v2))
              (handempty) (clear ?v1) (on ?v1 ?v2)))
```

Figure 1: STRIPS action schema coded in PDDL.

$\mathcal{L}(F)$. 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 $G \subseteq \mathcal{L}(F)$ 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* $\langle s_0, s_1, \ldots, s_n\rangle$ such that $s_0 = I$ and a_i ($1 \leq i \leq n$) is applicable in s_{i-1} and generates the successor state $s_i = \theta(s_{i-1}, a_i)$. 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 in the last state resulting from the application of the plan π in the initial state I.

The space of STRIPS models

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. We assume also that actions A are instantiated from a set of action schemas. Figure 1 shows the *stack* action schema, coded in PDDL, from a four-operator *blocksworld* (Slaney and Thiébaux 2001).

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 three-block four-operator *blocksworld* $\Omega = \{block_1, block_2, block_3\}$ while $\Omega_v = \{v_1, v_2\}$ because the operators with the maximum arity, `stack` and `unstack`, 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 eleven elements for the mentioned *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 define an STRIPS action schema as the tuple $\xi = \langle head(\xi), pre(\xi), add(\xi), del(\xi)\rangle$ where:

- $head(\xi) = \langle name(\xi), pars(\xi)\rangle$, is the *header* defined by its name and the corresponding *variable names*, $pars(\xi) = \{v_i\}_{i=1}^{ar(\xi)}$. The headers of a four-operator *blocksworld* are pickup(v_1), putdown(v_1), stack(v_1,v_2) and unstack(v_1,v_2).
- The preconditions $pre(\xi) \subseteq F_v$, the negative effects $del(\xi) \subseteq F_v$, and the positive effects $add(\xi) \subseteq F_v$ such that, $del(\xi) \subseteq pre(\xi)$, $del(\xi) \cap add(\xi) = \emptyset$ and $pre(\xi) \cap add(\xi) = \emptyset$.

For a given $head(\xi)$, then $F_v(\xi) \subseteq F_v$ defines the subset of elements that can appear in the preconditions and effects of the corresponding action schema. For instance, for the *stack* action schema $F_v(\mathtt{stack}) = F_v$ while $F_v(\mathtt{pickup})$={handempty, holding(v_1), clear(v_1), ontable(v_1), on(v_1,v_1)} excludes any element in F_v that involves v_2 because pickup(v_1) has arity one. The size of the space of possible STRIPS models for an action schema ξ is given by the expression, $2^{2|F_v(\xi)|}$. Note that STRIPS constraints require negative effects appearing as preconditions and that they cannot be positive effects and also, that positive effects cannot appear as preconditions. For the mentioned *blocksworld*, $2^{2|F_v(stack)|} = 4194304$ while $2^{2|F_v(pickup)|} = 1024$.

We say that two STRIPS operator schemes ξ and ξ' are *comparable* if both schemas share the same space of possible STRIPS models, i.e. iff $pars(\xi) = pars(\xi')$. For instance, we claim that blocksworld operators `stack` and `unstack` are *comparable* while `stack` and `pickup` 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 action schema $\xi' \in \mathcal{M}'$ and vice versa.

Model Recognition

Given $M = \{\mathcal{M}_1, \ldots, \mathcal{M}_m\}$, a finite and non-empty set of planning models and $\mathcal{T}$, the observation of a plan execution, *model recognition* is the task of identifying which model $\mathcal{M} \in M$ has the highest probability of producing $\mathcal{T}$. Our approach is to estimate this probability according to the amount of *edition* required by $\mathcal{M}$ to produce a plan execution that is compliant with $\mathcal{T}$.

This work assumes that models $\mathcal{M} \in M$ are *comparable* STRIPS planning models, so *edition* is focused on the action schemas $\xi \in \mathcal{M}$. In addition, the observation $\mathcal{T}$ is a sequence of *partially-observed* states and actions, $\mathcal{T} = \langle s_0, a_{,1}, s_1, \ldots, a_n, s_n\rangle$ s.t.:

- Intermediate states $s_i \in \mathcal{T}$, $1<i$, are partial assignment of values to fluents in which one or more literals can be missing, formally $0 \leq |s_i| \leq |F|$. In the extreme full intermediate states can be missing, i.e, $|s_i| = 0$.
- Intermediate actions a_i, $1 \leq i$, that transit between an observed state, s_i, to the next observed state, s_{i+1}, can be missing.

The conjunction of these two assumptions makes that transitions between two consecutive observed states may involve 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 the actual length of the plan whose execution is observed is unknown.

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

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 1. *Let $\mathcal{M}$ and $\mathcal{M}'$ be two* comparable STRIPS *action models. The* **edit distance***, denoted as $\delta(\mathcal{M}, \mathcal{M}')$, 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. In more detail, for an operator schema $\xi \in \mathcal{M}$ the maximum number of edits that can be introduced to their precondition set is $|F_v(\xi)|$ while the max number of edits that can be introduced to the effects is twice $|F_v(\xi)|$.

Definition 2. *The **maximum edit distance** of an* STRIPS *model $\mathcal{M}$ built from the set of possible elements F_v is $\delta(\mathcal{M}, *) = \sum_{\xi\in\mathcal{M}} 3 \times |F_v(\xi)|$.*

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

Definition 3. *Given $\mathcal{M}$, a* STRIPS *action model built from F_v and $\mathcal{T}$, an observation built with fluents in F and actions in A. The* **observation edit distance***, $\delta(\mathcal{M}, \mathcal{T})$, is the minimal edit distance from $\mathcal{M}$ to any* comparable *model $\mathcal{M}'$ s.t. $\mathcal{M}'$ produces a plan execution compliant with $\mathcal{T}$;*

$$\delta(\mathcal{M}, \mathcal{T}) = \min_{\forall \mathcal{M}' \rightarrow \mathcal{T}} \delta(\mathcal{M}, \mathcal{M}')$$

The STRIPS probability distribution

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 our scenario, hypotheses are about the comparable STRIPS models $\mathcal{M} \in M$ while the given observation is the partially observed plan execution $\mathcal{T}$.

We call the STRIPS *probability distribution*, denoted by $P(\mathcal{M}|\mathcal{T})$, to the probability distribution of models $\mathcal{M} \in M$ provided that the plan execution $\mathcal{T}$ is observed. This probability distribution can be estimated in three-steps:

1. Estimating the *a priori* probabilities $P(\mathcal{T})$ and $P(\mathcal{M})$. For instance given the $F_v(\xi)$ sets, if we assume that a priori all models are equiprobable $P(\mathcal{M}) = \frac{1}{\prod_{\xi\in\mathcal{M}} 2^{2|F_v(\xi)|}}$. With respect to the observation, if we assume that all the observations of plan executions with a maximum of n steps are equiprobable, $P(\mathcal{T}) = \frac{1}{2^{2n\times|F|\times|A|}}$.
2. Estimating the conditional probability $P(\mathcal{T}|\mathcal{M})$. For every possible model $\mathcal{M} \in M$, we can compute the *observation edit distance* $\delta(\mathcal{M}, \mathcal{T})$ and map it into a $[0, 1]$ likelihood with the expression, $1 - \frac{\delta(\mathcal{M},\mathcal{T})}{\delta(\mathcal{M},*)}$.
3. Applying the Bayes rule to obtain the normalized posterior probabilities (these probabilities must sum 1).

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

Model Recognition as Planning

For each $\mathcal{M} \in M$, we compute the *observation edit distance* $\delta(\mathcal{M}, \mathcal{T})$ using a classical planning compilation. 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 $\mathcal{M}'$ in the observation $\mathcal{T}$.** The solution plan continues with a postfix that validates the edited model $\mathcal{M}'$ on the given observations $\mathcal{T}$.

Figure 2 shows the plan for editing a given *blockswold* model where the positive effects `(handempty)` and `(clear ?v1)` of the `stack` schema are missing. The edited action model is validated at the observation of a four action plan for inverting a two-block tower where: states s_0 and s_4 are fully observed while the 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 deleitions) required by $\mathcal{M}'$ to be validated in the given observation, e.g. $\delta(\mathcal{M}, \mathcal{T}) = 2$ for the example in Figure 2. In this case $\delta(\mathcal{M}, *) = 3 \times 2 \times (11 + 5)$ since there are 4 action schemes (`pickup`, `putdown`, `stack` and `unstack`) and $|F_v| = |F_v(stack)| = |F_v(unstack)| = 11$ while $|F_v(pickup)| = |F_v(putdown)| = 5$.

Conditional effects

Conditional effects allow us to compactly define our compilation. 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 if and only if $\mathsf{pre}(a) \subseteq s$, 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 compilation formalization

Given a STRIPS model $\mathcal{M} \in M$ and the observation of a plan execution $\mathcal{T}$, our compilation outputs a classical planning task $P = \langle F, A, I, G \rangle$ such that:

- F contains:
  - The fluents built instantiating the predicates Ψ with the objects Ω that appear in any state/action observed in $\mathcal{T}$, i.e. the blocks `A` and `B` in the example of Figure 2.
  - Fluents $pre_f(\xi)$, $del_f(\xi)$ and $add_f(\xi)$, for every $f \in F_v(\xi)$, that represent the edited action model. If a fluent $pre_f(\xi)/del_f(\xi)/add_f(\xi)$ holds, it means that f is a precondition/negative/positive effect in the schema $\xi \in \mathcal{M}'$. For instance, the preconditions of the *stack* schema (Figure 1) are represented by the pair of fluents $\texttt{pre\_holding\_stack\_}v_1$ and $\texttt{pre\_clear\_stack\_}v_2$ set to True.
  - 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 $\mathcal{T}$. 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 $\mathcal{T}$.
  - The fluents $mode_{edit}$ and $mode_{val}$ to indicate whether the operator schemas are edited or validated, and the fluents $\{test_i\}_{1 \leq i \leq n}$, indicating the state observation $s_i \in \mathcal{T}$ where the action model is validated.
- I encodes the first state observation, $s_0 \subseteq F$ and sets to true $mode_{edit}$ 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 $\mathcal{M}$.
- $G = \bigcup_{1 \leq i \leq n} \{test_i\}$, requires that the edited model is validated in the state observations $s_i \in \mathcal{T}$.
- A comprises three kinds of actions with conditional effects:

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

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

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

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

   Besides these actions, A also contains the actions for *inserting* a precondition and for *deleting* 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 3 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 $\mathcal{T}$ includes observed actions, then 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 in $\mathcal{T}$.

3. Actions for *validating* the partially observed state $s_i \in$

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

	q_0	q_1	q_2	q_3	q_4	$\underline{q_5}$
a	x,r,q_1	a,r,q_1	-	a,l,q_3	-	-
b	-	y,r,q_2	b,r,q_2	b,l,q_3	-	-
c	-	-	z,l,q_3	-	-	-
x	-	-	-	x,r,q_0	-	-
y	y,r,q_4	y,r,q_1	-	y,l,q_3	y,r,q_4	-
z	-	-	z,r,q_2	z,l,q_3	z,r,q_4	-
$\Box$	-	-	-	-	$\Box$,r,q_5	-

Figure 4: Seven-symbol six-state *Turing Machine* for recognizing the $\{a^n b^n c^n : n \geq 1\}$ language ($\underline{q_5}$ is the only acceptor state).

$\mathcal{T}, 1 \leq i < n$.

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

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*.

Modeling *Turing Machines* with STRIPS

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 such that $q_0 \in Q$ is the initial state of the machine and $Q_\perp \subseteq Q$ is the subset of acceptor states.
- Σ is the *tape alphabet*, that is a finite non-empty set of symbols that contains the *input alphabet* $\Upsilon \subseteq \Sigma$ (the subset of 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).
- $\delta : \Sigma \times (Q \setminus Q_\perp) \rightarrow \Sigma \times Q \times \{left, right\}$ is the *transition function*. For each possible 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 4 shows the δ function of a *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 possible 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.

A classical planning frame $\Phi = \langle F, A \rangle$ can encode the *transition function* δ of a *Turing Machine* $\mathcal{M}$ as follows:

- Fluents F are instantiated from a set of four *predicates* Ψ: `(head ?x)` that encodes the current position of the header in the tape. `(next ?x1 ?x2)` encoding that the cell `?x2` follows cell `?x1` in the tape. `(symbol-`σ `?x)` encoding that the tape cell `?x` contains the symbol $\sigma \in \Sigma$. `(state-`q`)` encoding that $q \in Q$ is the current machine state. Given a set of *objects* Ω that represent the cells in the tape of the given Turing Machine, the set of fluents F is induced by assigning objects in Ω to the arguments of the predicates in Ψ.
- Actions A are instantiated from STRIPS operator schema. For each transition in δ, a STRIPS schema is defined:
  - The **header** is `transition-id(?xl ?x ?xr)` where id uniquely identifies the transition in δ. Parameters $?xl$, $?x$ and $?xr$ are tape cells.
  - The **preconditions** are `(head ?x)` and `(next ?xl ?x) (next ?x ?xr)` to make $?x$ the tape cell pointed by the header and $?xl/?xr$ its left/right neighbours. Preconditions `(symbol-`σ `?x)` and `(state-`q`)` are also included to capture the symbol currently pointed by the header and the current machine state.
  - The **delete effects** remove the symbol currently pointed by the header and the currrent machine state while the **positive effects** set the new symbol pointed by the header and the new machine state according to δ.

```
(: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 5: STRIPS action schema that models the transition $a, q_0 \rightarrow x, r, q_1$ of the Turing Machine defined in Figure 4.

The STRIPS action schema of Figure 5 models the rule $a, q_0 \rightarrow x, r, q_1$ of the *Turing Machine* defined in Figure 4 (the full encoding of the *Turing Machine* of Figure 4 produces a total of sixteen STRIPS action schema).

Since the *transition function* of a *Turing Machine* can be encoded as a classical planning frame $\Phi = \langle F, A \rangle$, executions of that *Turing Machine* are definable as *plan traces* $\mathcal{T} = \langle s_0, a_1, s_1, \ldots, a_n, s_n \rangle$.

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*. We randomly choose a machine $\mathcal{M} \in M$ and produce an fifty-step execution plan trace $\mathcal{T} = \langle s_0, a_1, s_1, \ldots, a_{50}, s_{50} \rangle$. Finally, we follow our *model recognition as planning* method to identify the *Turing Machine* that produced $\mathcal{T}$. This exeperiment is repeated for different amounts of missing information in the input trace $\mathcal{T}$: 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.

Results

Conclussions

Our $\delta(\mathcal{M}, \mathcal{T})$ 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 $\mathcal{T}$ 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 normally smaller than the number of objects in the observed states.

The STRIPS *probability distribution* assigns the same probabaility value to any hypothesis model $\mathcal{M} \in M$ that can reproduce the given observation $\mathcal{T}$. As a rule of thumb, the less information observed the more different models can reproduce observations. If we assume that the observed agent is acting rationally, like in *plan recognition as planning* (Ramírez 2012; Ramírez and Geffner 2009), one can define probability distributions that consider as more probable hypothesis the ones that reproduce observations in less steps. A related approach is recently followed in *model reconciliation* (Chakraborti et al. 2017) where model edition is used to conform the different 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*.

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.

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.