AMLSI: A Novel Accurate Action Model Learning Algorithm

Maxence Grand, Damien Pellier, Humbert Fiorino

AMLSI: A Novel Accurate Action Model Learning Algorithm

Maxence Grand, Damien Pellier and **Humbert Fiorino**
Laboratoire d'Informatique de Grenoble - MARVIN
Université Grenoble Alpes, Saint Martin d'Hères, France
{Maxence.Grand, Damien.Pellier, Humbert.Fiorino}@univ-grenoble-alpes.fr

Abstract

This paper presents a new algorithm based on grammar induction, called AMLSI (**A**ction **M**odel **L**earning with **S**tate machine **I**nteractions), to learn PDDL domains by querying the system to model with action sequences and by observing the state transitions. AMLSI takes as input a training set of feasible and infeasible action sequences built from partial and noisy observations and returns a PDDL domain. A key issue for domain learning is the ability to plan with the learned domains. It often happens that a small learning error leads to a domain that is unusable for planning. Unlike other algorithms, we show that AMLSI is able to lift this lock by learning domains from partial and noisy observations with sufficient accuracy to allow planners to solve new problems.

1 Introduction

Hand-encoding PDDL (*Planning Domain Description Language*) domains (McDermott *et al.* 1998) is generally considered as difficult, tedious and error-prone. The experts of the system to model are not always PDDL experts and vice versa. In order to tackle this issue, planning domain learning algorithms have been proposed to automatically generate PDDL domains, e.g., (Yang *et al.* 2007; Shahaf and Amir 2006; Mourão *et al.* 2012; Cresswell *et al.* 2009; Cresswell and Gregory 2011; Gregory and Cresswell 2015). A challenging issue for these learning algorithms is their ability to generate domains that planners can use to solve new planning problems. In practice, most state of the art approaches do not evaluate this ability. They use rather the syntactical error (differences in the preconditions and the effects of operators) between International Planning Competition (IPC) benchmarks and learned domains. Unfortunately, it is not because one learned domain is syntactically closer to a IPC domain that it is better than another. It often happens that a small syntactical error leads to a domain that is unusable for planning: the ability of a PDDL domain to solve planning problems can depend on a few number of decisive preconditions/effects.

In this paper, we propose a new algorithm based on grammar induction, called AMLSI (**A**ction **M**odel **L**earning with **S**tate machine **I**nteractions), to learn PDDL domains by querying the system to model with action sequences and observing the state transitions. AMLSI takes as input a training set of feasible and infeasible action sequences built from partial and noisy observations and returns a PDDL domain. Unlike other approaches, we show that AMLSI is able to learn domains from partial and noisy observations with sufficient accuracy to allow planners to solve new problems.

The rest of the paper is organized as follows. In section 2 we present the related works. In section 3 we present the problem statement and, in section 4, we detail the AMLSI algorithm. Finally, section 5 evaluates the performance of AMLSI on IPC benchmarks.

2 Related Works

Many approaches have been proposed to learn planning domains. Table 1 proposes a synthesis of these works, which can be classified according to the input data of the learning process and the complexity of the language used in the output domain. The input data can be plan traces obtained by resolving a set of planning problems, e.g, ARMS (Yang *et al.* 2007), plan traces and partial planning domains to complete, e.g, EXPO (Gil 1994) and more recently RIM (Zhuo *et al.* 2013), or random walks, e.g., IRaLe (Rodrigues *et al.* 2010). The input data can be complete (states and actions), e.g., Observer (Wang 1995) or completely blind, e.g., LOCM (Cresswell *et al.* 2009), LOP (Gregory and Cresswell 2015) or NLOCM (Gregory and Lindsay 2016), noisy, e.g., PlanMiner (Segura-Muros *et al.* 2018) or noiseless, e.g., Louga (Kucera and Barták 2018). As output, the learned planning domains can have different levels of expressivity: negative preconditions, e.g., Louga (Kucera and Barták 2018), static relations, e.g., LSO-NIO (Mourão *et al.* 2012), numerical functions, e.g., LOCM2 (Cresswell and Gregory 2011) or conditional effects, e.g., SLAF (Shahaf and Amir 2006).

AMLSI differs from the state-of-the-art algorithms in several points. Firstly, AMLSI is one of the few algorithms able to deal with noisy and partial observations. Secondly, AMLSI works with both feasible and infeasible action sequences while most approaches use only feasible action sequences or plan traces. To our best of knowledge, only IRALe and LSO-NIO use failures in action sequences, but AMLSI differentiates feasible and infeasible action sequences. In addition, AMLSI uses random walks to generate its training datasets of both feasible and infeasible ac-

	Input	Observations	Noise	Negative preconditions	Static relations	Numerical functions	Conditional effects
EXPO	Partial domain, Plan traces	Complete	No	NA	NA	No	No
RIM	Partial domain, Plan traces	Complete	No	NA	NA	No	No
Observer	Plan traces	Complete	No	No	NA	No	No
ARMS	Plan traces	Partial	No	NA	NA	No	No
SLAF	Plan traces	Partial	No	NA	No	No	Yes
Louga	Plan traces	Partial	No	Yes	Yes	No	No
Plan-Milner	Plan traces	Partial	Yes	NA	No	Yes	No
LOCM	Plan traces	No Observation	-	NA	No	No	No
LOCM2	Plan traces	No Observation	-	NA	No	No	No
LOP	Plan traces	No Observation	-	NA	Yes	No	No
NLOCM	Plan traces	No Observation	-	NA	Yes	Yes	No
IRALe	Random walk	Complete	Yes	NA	NA	No	No
LSO-NIO	Random walk	Partial	Yes	No	Yes	No	No
AMLSI	Random walk	Partial	Yes	Yes	Yes	No	No

Table 1: State-of-the-art action model learning algorithms. For some algorithms, there is no indication on the ability of the algorithm to learn static relations and/or negative preconditions. These algorithms are rated NA for these criteria.

tion sequences, whereas most of the algorithms like ARMS, Louga and Plan-Milner either only use plan traces or random walks generating feasible action sequences. Thirdly, in terms of expressivity, AMLSI learns PDDL domains including static relations in preconditions as well as negative preconditions. To our best knowledge, only Louga has the same expressivity but it cannot work with noisy traces. Fourthly, AMLSI uses automata representations like LOCM though differently: LOCM generates several automata, one for each domain object while AMLSI generates a single automaton representing the whole domain for a given initial state. The advantage of this method is that we can use regular grammar induction algorithms, which is a well-defined area with accurate algorithms. More importantly, AMLSI is the only algorithm able to learn planning domains accurate enough to be used by planners to solve new planning problems (i.e. that are not in the training sets) with such a level of noise in the observations (see Section 5).

3 Problem Statement

We work in the context of classical planning. World states s are modeled as sets of propositions and actions change the world states. For instance, in the gripper domain,

$$\begin{aligned} s = \quad & (at - robby\ r_1) \wedge \neg(at - robby\ r_2) \wedge (at\ b\ r_1) \wedge \\ & \neg(at\ b\ r_2) \wedge (free\ left) \wedge (free\ right) \wedge \\ & \neg(carry\ b\ right) \wedge \neg(carry\ b\ left) \end{aligned}$$

states that $robby$ is in the room $r1$ as well as the ball b, and $robby$'s arms are both $free$.

Formally, let S be a set of all the propositions modeling properties of world, and A the set of all the possible actions in this world. A state s is a subset of S and each action $a \in A$ is a triple of proposition sets $(\rho_a, \epsilon_a^+, \epsilon_a^-)$, where $\rho_a, \epsilon_a^+, \epsilon_a^- \subseteq S$, and $\epsilon_a^+ \cap \epsilon_a^- = \emptyset$. ρ_a are the preconditions of action a, that is, the propositions that must be in the state before the execution of action a. ϵ_a^+ and ϵ_a^- are respectively the positive (add list) and the negative (del list) effects of action a, that is, the propositions that must be added or deleted in s after the execution of the action a. In practise, actions are abstracted as operators in first order logic Planning Domain Description Language. For instance $move(?from\ ?to)$ stands for all the possible groundings of the parameters $?from$ and $?to$ and the actions $move(r_1\ r_2)$, $move(r_2\ r_1)$ etc.

Let $\gamma : S \times A \to S$ be the state transition function such that $s' = \gamma(s, a) = (s \cup \epsilon_a^+) \setminus \epsilon_a^-$. $\gamma(q, a)$ is defined if and only if $\rho_a \subseteq s$. Let $\pi = [a_0, a_1, \ldots, a_n]$ be a sequence of actions, and $+$ the concatenation of two sequences. $\Gamma(s_0, \pi)$ is defined recursively as follows:

$$\Gamma(s_0, \pi) = \begin{cases} [s_0] & \text{if } \pi = [] \\ [s_0] & \text{if } \rho_{a_0} \not\subseteq s_0 \\ [s_0] + \Gamma(\gamma(s_0, a_0), [a_1, \ldots, a_n]) & \text{otherwise} \end{cases}$$

In this work, we assume that:

- the set of all the possible actions A is known (for instance $A = \{move(r_1\ r_2), move(r_2\ r_1), pick(b\ r_1\ grip), pick(b\ r_2\ grip), drop(b\ r_1\ grip), drop(b\ r_2\ grip)\}$) and for all $a \in A$, $(\rho_a, \epsilon_a^+, \epsilon_a^-)$ is unknown,
- γ is the state machine to model and Γ represents the observed states: if a query (s_0, π) is sent to the state machine, it sends back a sequence of observable states $\Gamma(s_0, \pi)$. This kind of state machines is, for instance, an ATM on which a sequence of actions (inserting a credit card, a PIN number etc.) is tested, any process computing streams of user commands, an industrial process simulator or a person who is interviewed ("what if in this state, we perform such and such action?") to collect the expertise on such a process etc.
- the observations $\Gamma(s_0, \pi)$ are possibly partial and noisy. A partial observation is a state where some propositions are missing. For instance, in the gripper domain,

  $$\begin{aligned} & (at - robby\ r_1) \wedge \ldots \wedge (at\ b\ r_1) \wedge \\ & \neg(at\ b\ r_2) \wedge \ldots \wedge (free\ right) \wedge \\ & \neg(carry\ b\ right) \wedge \neg(carry\ b\ left) \end{aligned}$$

  And in a noisy observation, the truthiness of some predicates is incorrect:

  $$\begin{aligned} & (at - robby\ r_1) \wedge \ldots \wedge (at\ b\ r_1) \wedge \\ & \neg(at\ b\ r_2) \wedge \ldots \wedge (free\ right) \wedge \\ & (\mathbf{carry\ b\ right}) \wedge \\ & (\mathbf{carry\ b\ left}) \end{aligned}$$

Algorithm 1: AMLSI

input : A: Set of actions, P: Set of propositions, Sys: A System
output : D: A PDDL domain

1 $I_+, I_- \leftarrow generation(A, Sys)$;
2 $I_+^P, I_-^P \leftarrow preprocessing(I_+, I_-, A)$;
3 $Aut \leftarrow RPNI(I_+^P, I_-^P)$;
4 $\mu_A, \mu_P \leftarrow mapping(Aut, I_+)$;
5 $D \leftarrow generateOperator(Aut, \mu_A, \mu_P)$;
6 **repeat**
7 $\quad D' \leftarrow D$;
8 $\quad$ **repeat**
9 $\quad\quad D'' \leftarrow D$;
10 $\quad\quad D \leftarrow effectRefinement(D, Aut, \mu_A, \mu_P)$;
11 $\quad\quad D \leftarrow preconditionRefinement(D)$;
12 $\quad$ **until** $D = D''$;
13 $\quad D \leftarrow tabuSearch(I_+, I_-, D)$;
14 **until** $D = D'$;
15 **return** D

$\pi = [a_0, a_1, \ldots, a_n]$ is a *feasible* sequence given s_0 if and only if $\Gamma(s_0, [a_0, a_1, \ldots, a_n]) = [s_0, s_1, \ldots, s_n]$: both sequences have the same length. Therefore, $\pi = [a_0, a_1, \ldots, a_n]$ is an *infeasible* sequence given s_0 if and only if $\Gamma(s_0, [a_0, a_1, \ldots, a_n]) = [s_0, s_1, \ldots, s_i]$ and $i < n$. Or, equivalently, $[a_0, a_1, \ldots, a_{i-1}]$ is feasible and $[a_0, a_1, \ldots, a_{i-1}, a_i]$ is infeasible (given s_0).

In this paper we tackle the following problem: *by repeatedly querying a state machine with different (possibly infeasible)* (s_0, π) *and observing* $\Gamma(s_0, \pi)$ *(possibly partial and noisy), is it possible to learn the state transition function* γ *and express it as a PDDL domain?*

4 The AMLSI approach

The AMLSI approach, resumed by algorithm 1, is composed of four steps: (1 - line 1) a dataset generation step, (2 - lines 2-3) a grammar induction step, (3 - lines 4-5) a PDDL induction step and (4 - lines 6-14) a refinement step.

Step 1. Dataset generation

The objective of this step is to built up two training datasets: I_+ (positive samples) containing the feasible action sequences and the corresponding observations, and I_- (negative samples) containing the infeasible action sequences: at a given state s, we query the system about the feasibility of an action a randomly chosen in A. If a is feasible, the current state is observed and we add a to the current π. This random walk is iterated until π reaches an arbitrary length, and added to I_+. If a is infeasible in the current state, the concatenation of π and a is added to I_-.

Step 2. Grammar Induction

In the second step, we use the RPNI algorithm (Oncina and García 1992) to learn the regular grammar based on I_+ and I_- inputs. RPNI has a polynomial complexity and is optimal: it returns the smallest automaton (a regular grammar can be represented as a deterministic finite automaton) accepting all the positive samples I_+ and rejecting all the negative samples $I-$ when I_+ and $I-$ are "characteristic" (Oncina and García 1992). Formally, the deterministic finite automaton learned is a quintuple $< A, N, n_0, \gamma, F >$, where A is the set of action, N is the set of nodes, $n_0 \in N$ is the initial node, γ is the node transition function, and $F \subseteq N$ is the set of final nodes.

Finally, to prepare the grammar induction, we carry out two preprocessing steps for I_+ and I_-. These two preprocessing steps aim to extend I_+ and I_-:[1]

Step 1. Sequence prefixes. Any prefix of a feasible sequence is itself feasible. The correct grammar has to accept them, so they are added to I_+. For instance, if $[move(r_1\ r_2), move(r_2\ r_1), pick(b\ r_1\ left)] \in I_+$, $[move(r_1\ r_2), move(r_2\ r_1)]$ and $[move(r_1\ r_2)]$ are also in I_+.

Step 2. Pairwise constraints. Pairwise constraints are pairs of actions that cannot be consecutive in a sequence. These constraints are based on the fact that for an action to be feasible a certain number of resources must be produced (add list) and others must be consumed (del list). For instance, in the gripper domain, $move(r_1\ r_2)$ is never followed by $pick(b\ r_1\ grip)$ because $(at - robby\ r_1)$ must be true to execute $pick(b\ r_1\ grip)$, and after the execution of $move(r_1\ r_2)$, $(at - robby\ r_1)$ is always false, therefore $pick(b\ r_1\ grip)$ cannot follow $move(r_1\ r_2)$. Pairwise constraint computation is based on I_+: we assume that only pairs of actions in I_+ are possible. Therefore, every pair of actions that is not in a sequence of I_+ is added to I_-. Formally, the set of pairwise constraints is built as follows:

$$\{\forall a_i, a_j \in A^2 | \nexists \pi \in I_+ \ s.t. \ \pi = \pi_1, a_i, a_j, \pi_2\}$$

Step 3. Operator generation

Operator generation is based on four steps:

1. Observation mapping Once the automaton is induced, we need to know which node of the automaton correspond to which observed state. To do that, we input in the automaton all the positive samples in I_+ and map the pairs "node, action" in the automaton with the pairs "state, action" in I_+. There are two different mapping: the mapping (A)nte μ_A and the mapping (P)ost μ_P. $\mu_A(n, a)$ (resp. $\mu_P(n, a)$) gives the state before (resp. after) the execution of the transition a in node n. Then, we compute a reduced mapping $\bigcap \mu_A$ (resp $\bigcap \mu_P$), which contains the common propositions of all the states for a given (n, a). For instance, consider the classical gripper planning domain: $(at\ b\ r_1)$ is in $\bigcap \mu_A(0, pick(b\ r_1\ grip))$ iff $(at\ b\ r_1)$ is in all $\mu_A(0, pick(b\ r_1\ grip))$. Likewise, we remove from $\bigcap \mu_A(0, pick(b\ r_1\ grip))$ and $\bigcap \mu_P(0, pick(b\ r_1\ grip))$ all the propositions whose parameters are not a subset of

[1] Note that these preprocessing are only used for the grammar induction step.

Algorithm 2: Precondition generation

input : o: an operator
output : ρ_o: preconditions of o

1 $\Xi \leftarrow \{\}$;
2 **for** *a instance of o* **do**
3 $\quad \tau \leftarrow \{\}$;
4 $\quad$ **for** *n in the automaton* **do**
5 $\quad\quad$ **if** *a is an outgoing edge of n* **then**
6 $\quad\quad\quad$ **if** $\bigcap \mu_A(n,a) \neq \emptyset$ **then**
7 $\quad\quad\quad\quad \tau \leftarrow \tau \cup \{\bigcap \mu_A(n,a)\}$;
8 $\quad\quad\quad$ **end**
9 $\quad\quad$ **end**
10 $\quad$ **end**
11 $\quad \rho_a \leftarrow \bigcap(\tau)$;
12 $\quad \Xi \leftarrow \Xi \cup \{generalize(\rho_a)\}$;
13 **end**
14 $\rho_o \leftarrow \bigcap(\Xi)$;
15 **return** ρ_o

$(b\ r_1\ grip)$, the parameters of $pick(b\ r_1\ grip)$ (classical planning assumption (Fikes and Nilsson 1971)).

The advantage to use the common pattern in the mapping is that it reduces the noise in the mapping. Indeed, when we have a high level of compression, it is unlikely that a proposition present in the mapping is a misjudged proposition.

2. Precondition generation The algorithm 2 describes how we generate operator preconditions. To learn the preconditions of an operator o, we compute the reduced mapping of all the nodes where an action a, instantiating o, is feasible. At line 8, ρ_a is the reduced mapping of all the μ_A mappings of the action a. The generalization of ρ_a at line 9 replaces all the constant symbols by variables (for instance, $(b\ r_1\ grip)$ by $(?obj\ ?room\ ?gripper)$).

3. Effect generation Effect generation is analogous to precondition generation with the difference that we use $\mu_A(n,a)$ and $\mu_P(n,a)$ to compute a reduced mapping of ϵ_a^+, the propositions that are never in $\mu_A(n,a)$ and always in $\mu_P(n,a)$, and ϵ_a^-, the propositions that are always in $\mu_A(n,a)$ and never in $\mu_P(n,a)$.

Step 4. Operator refinement

The refinement steps of the PDDL operators are necessary because the observations are partial and noisy. The refinement is divided into 3 substeps:

1. Effect refinement This step ensures that the generated operators allow to regenerate the induced grammar. We use the reduced mappings $\bigcap \mu_A$ to verify that for each couple of consecutive actions a and a', the effects of the action a generate the preconditions of action a'. If it is not the case, we add in the effects of a the propositions satisfying the preconditions of a'. For instance, suppose we have $n' = \gamma(n, move(r_1\ r_2))$ and $n'' = \gamma(n', pick(b\ r_2\ grip))$. Now suppose we have $\neg(at - robby\ r_2) \in \bigcap \mu_A(n, move(r_1\ r_2))$, $(at - robby\ r_2) \in \rho_{pick(b\ r_2\ grip)}$ and $(at - robby\ r_2) \notin \epsilon^+_{move(r_1\ r_2)}$. We need to have $(at - robby\ r_2) \in \epsilon^+_{move(r_1\ r_2)}$ in order to make $\gamma(n, move(r_1\ r_2))$ and $\gamma(n', pick(b\ r_2\ grip))$ feasible. Thus, we add $(at - robby\ ?to)$ to $\epsilon^+_{move(?from\ ?to)}$.

2. Precondition refinement In this step, we assume like (Yang *et al.* 2007) that the propositions of the negative effects must be in the action preconditions. Thus for each negative effects in an operator, we add the corresponding proposition in the preconditions. For instance, suppose $\neg(at - robby\ ?from) \in \epsilon^-_{move(?from\ ?to)}$, then $(at - robby\ ?from) \in \rho_{move(?from\ ?to)}$ after refinement.

Since effect refinement depends on the preconditions and precondition refinement depends on the effects, we repeat these two steps until convergence. They converge because the adding of preconditions is limited by the effects, and the adding of effects is limited by the preconditions of the next action in the induced automaton (In our experiments, see Section 5, less than 10 iterations are needed to converge).

3. Tabu search Finally, we perform a tabu search to improve the PDDL operators independently of the induced grammar, on which operator generation is based.

. The neighborhood of a candidate domain is the set of domains where a precondition or an effect is added or removed. And the search space of the tabu search is the set of all possible domains compatible with the PDDL syntax constraints (Yang *et al.* 2007). The fitness function used to evaluate a candidate set D of PDDL operators is:

$$J(D|I_+, I_-) = \begin{array}{l} J_\rho(D|I_+) + J_\epsilon(D|I_+) + \\ J^+(D|I_-) + J^-(D|I_-) \end{array}$$

where :

- $J_\rho(D|I_+) = \sum_{\pi \in I_+} \sum_{s \in \Gamma(s_0,\pi)} Accept(\rho_a, s) - Reject(\rho_a, s)$
  computes the fitness score for the preconditions of the actions a. $Accept(\rho_a, s)$ counts the number of positive and negative preconditions in the observed state s, and $Reject(\rho_a, s)$ counts the number of positive and negative preconditions that are not in s.
- $J_\epsilon(D|I_+) = \sum_{\pi \in I_+} \sum_{s \in \Gamma(s_0,\pi)} Equal(s, \hat{s}) - Different(s, \hat{s})$
  computes the fitness score for the effects of the actions a. s are the observed states and $\hat{s}$ are the states obtained by applying the actions of π. $Equal(s, \hat{s})$ counts the number of similar propositions in s and $\hat{s}$, and $Different(s, \hat{s})$ counts the differences.
- $J^+(D|I_+) = \sum_{\pi \in I_+} |\pi| \times \mathbb{1}_{Accept(D,\pi)}$ where $\mathbb{1}_{Accept(D,\pi)} = 1$ if and only if D can generate the positive sample π. $|\pi|$ is the length of π.
- $J^-(D|I_-) = \sum_{\pi \in I_-} \mathbb{1}_{Accept(D,\pi_+) \wedge Reject(D,\pi)}$. As detailed in section 15, the negative sample $\pi \in I_-$ is a sequence of $n+1$ actions where the n first actions are a prefix of a sequence in I_+: π_+ is this prefix. $\mathbb{1}_{Accept(D,\pi_+) \wedge Reject(D,\pi)} = 1$ if and only if D can generate π_+ and not π.

Fitness functions $J_\rho(D|I_+)$ and $J_\epsilon(D|I_+)$ measure the ability of D to explain the observed states whereas fitness functions $J^+(D|I_+)$ and $J^-(D|I_-)$ measure its ability to regenerate the induced grammar. $J^+(D|I_+)$ is weighted by the length of $\pi \in I_+$ but not $J^-(D|I_-)$ because I_- is larger than I_+.

Once tabu search is done, we repeat all the refinement steps until convergence.

5 Experiments

Our experiments[2] are based on 10 IPC domains: Blocksworld, Gripper, Hanoi, N-Puzzle, Peg Solitaire, Parking and Zenotravel, Sokoban, Neg-Visit-All and Neg-Elevator. Neg-Visit-All and Neg-Elevator are modified versions of Visit-All and Elevator with negative preconditions to show AMLSI ability to learn them. All the used benchmarks are STRIPS domain. Table 2 shows our experimental setup.

We deliberately chose the size of the test sets larger than the learning sets to show AMLSI's ability to learn accurate domains with small datasets. The training and test sets are generated as explained in Section 15. In the training sets, we generate positive action sequences with a length randomly chosen between 10 and 20, and in the test sets, we generate positive action sequences with a length randomly chosen between 1 and 100. I_- are bigger than I_+ because it is more likely to generate infeasible actions.

We test each IPC domain with 3 different initial states over five runs, and we used five seeds randomly generated for each run. Tabu search is computed over 200 runs. For each IPC domain, we generate observed states by randomly removing a fraction of the propositions (partial states) and by randomly modifying their truth values. All tests were performed on an Ubuntu 14.04 server with a multi-core Intel Xeon CPU E5-2630 clocked at 2.30 GHz with 16GB of memory.

5.1 Evaluation Metrics

AMLSI is evaluated with 4 metrics of the literature:

Syntactical error The syntactical error $error(o)$ (Zhuo *et al.* 2010) for an operator o is defined as the number of extra or missing predicates in the preconditions[3] ρ_o, the positive effects ϵ_o^+ and the negative effects ϵ_o^- divided by the total number of possible predicates. The syntactical error for a domain with a set of operator O is: $E_\sigma = \frac{1}{|O|}\sum_{o\in O} error(o)$.

Precondition error rate The precondition error rate (Yang *et al.* 2007) computes the rate of preconditions that are not satisfied in the positive test set. This metric measures the quality of the preconditions in the learned domain. It is computed as follows:

$$E_\rho = \sum_{\pi\in E^+} \frac{\sum\limits_{a\in\pi} error(\rho_a, \hat{s})}{\sum\limits_{a\in\pi} |\rho_a|}$$

$error(\rho_a, \hat{s}) = |\{p \in \rho_a \wedge \neg p \in \hat{s}\}| + |\{\neg p \in \rho_a \wedge p \in \hat{s}\}|$ gives the number of positive and negative preconditions p in actions a that are not satisfied in the observed state $\hat{s}$.

Effect error rate The effect error rate (Kucera and Barták 2018) computes the error rate on the effects of the learned domain. It is computed as follows:

$$E_\epsilon = \sum_{\pi\in E^+} \frac{\sum\limits_{a\in\pi} error(\epsilon_a, \hat{s})}{\sum\limits_{a\in\pi} |\epsilon_a|}$$

$error(\epsilon_a, \hat{s}) = |\{p \in \epsilon_a^+ \wedge \neg p \in \hat{s}\}| + |\{\neg p \in \epsilon_a^- \wedge p \in \hat{s}\}|$ gives the number of positive and negative effects p in the actions a that are not satisfied in the observed state $\hat{s}$.

Accuracy The accuracy (Zhuo *et al.* 2013) quantifies the usability of the learned model for planning. Most of the works addressing the problem of learning planning domains uses the syntactical error to quantify the performance of the learning algorithm. However, domains are learned to be used for planning, and it often happens that one missing precondition or effect makes them unable to solve new planning problems. Formally, the accuracy $Acc = \frac{N}{N^*}$ is the ratio between N the number of correctly solved problems with the learned domain and N^* the total number of problems to solve. The accuracy is computed over 20 problems. Although rarely used, we argue that the accuracy is the most important metric because it measures to what extent a learned domain is useful in practise for planning. Problem are solved with Fast Downward 19.06 (Helmert 2006). Plan validation is realized with the automatic validation tool used in the IPC competition: VAL (Howey and Long 2003). Finally, we also report in our results the ratio of solved problem that are not necessarily correctly solved.

5.2 Results

In order to study the performances of AMLSI with respect to noisy and partial observations, we use three different experimental scenarios:

1. Complete intermediate observations (100%) and no noise (0%),
2. Complete intermediate observations (100%) and high level of noise (20%),
3. Partial intermediate observations (25%) and no noise (0%).
4. Partial intermediate observations (25%) and high level of noise (20%).

Results are given by the Table - 3.

[2] AMLSI and our experimental setup can be found in an anonymous dropbox: https://www.dropbox.com/sh/3gpcdasycuy4r9h/AACYDrwNeseOMFLX6tlmMZcWa?dl=0

[3] Note that we compute the syntactical error without taking into account the negative preconditions of the operators because some of the chosen IPC domains do not have them.

Domain	$\#Operators$	$\#Predicates$	$\#Objects$	$\#Actions$	$\#Propositions$	$\lvert I_+\rvert$	$\lvert I_-\rvert$	$\lvert\pi_+\rvert$	$\lvert\pi_+\rvert$	$\lvert E_+\rvert$	$\lvert E_-\rvert$	$\lvert e_+\rvert$	$\lvert e_+\rvert$
Blocksworld	4	5	3	18	16	30	2421.3	15	8.3	100	26209.5	49	33.6
Gripper	3	4	5	10	8	30	1168.1	15.2	8.3	100	12940.3	50.7	33.7
Hanoi	4	7	6	24	25	30	3011.3	14.8	8.2	100	34780.7	50.6	33.8
N-Puzzle	1	3	7	24	24	30	3354.1	15.2	8.4	100	36613.4	49.9	34
Peg Solitaire	3	5	9	32	39	30	3465.2	6.9	5.4	100	11171.9	6.5	5.3
Parking	4	5	6	60	24	30	5729.53	15	8.5	100	65216.6	50.6	34
Zenotravel	5	5	7	14	10	30	1631.4	15.1	8.4	100	17850.3	49.6547	33.9
Sokoban	2	4	14	36	51	30	4634.33	15	8.2	100	51832.7	51.3	33.4
Neg Visit All	2	3	4	9	13	30	1275.7	15	8.9	100	16666.1	51	35.6
Neg Elevator	4	6	5	8	13	30	1050.7	15.1	8.6	100	13086.6	51	35.7

Table 2: Benchmark domain characteristics (from left to right): number of operators, number of predicates, number of objects in each initial states, number of actions in each initial states, number of propositions in each initial states, average size of $|I_+|$ and $|I_-|$ training sets, the average length of the positive (resp. negative) training sequences $\pi_+ \in I_+$ (resp. $\pi_- \in I_-$), average size of $|E_+|$ and $|E_-|$ test sets, the average length of the positive (resp. negative) test sequences $e_+ \in E_+$ (resp. $e_- \in E_-$).

Fluents		100%										25%									
Noise		0%					20%					0%					20%				
Domain	Algorithm	$E_\rho(\%)$	$E_\epsilon(\%)$	$E_\sigma(\%)$	$Solved(\%)$	$Acc(\%)$	$E_\rho(\%)$	$E_\epsilon(\%)$	$E_\sigma(\%)$	$Solved(\%)$	$Acc(\%)$	$E_\rho(\%)$	$E_\epsilon(\%)$	$E_\sigma(\%)$	$Solved(\%)$	$Acc(\%)$	$E_\rho(\%)$	$E_\epsilon(\%)$	$E_\sigma(\%)$	$Solved(\%)$	$Acc(\%)$
Blocksworld	AMLSI	0	0	0	100	100	0.8	0.8	0.6	93.7	93.7	0	0	0	100	100	2.1	1.5	1.2	76.3	76.3
	Generation step	0	0	0	100	100	0	0	33.25	0	0	0	0	19.1	0	0	0	0	35.8	0	0
	Simple refinement	0	0	0	100	100	7.7	6.6	26.3	20	0	0	0	0	100	100	0	0	28.7	20	0
	Tabu search alone	0	0	9.8	86.7	13	0.3	0.5	9.6	87	13	2.34	4.5	12.3	61.7	9	2.4	4.7	13.3	67	10.3
	Without PC	19.5	29.1	22.4	27.7	27.7	12.3	21.3	18.5	53.3	34.3	19.2	29.8	22.1	27.7	27.7	13.3	21.8	16.7	60	36.3
	LSO-NIO	0	0	0	100	100	13.5	12.9	20.1	0	0	0	0	28.1	26.7	0	15.9	18.8	33.4	40.3	0
Gripper	AMLSI	0	0	0	100	100	0	0	0	100	100	0	0	0	100	100	0	0	0	100	100
	Generation step	0	0	0	100	100	0	0	46.9	0	0	0	0	0.9	86.7	86.7	0	0	37.6	0.7	0
	Simple refinement	0	0	0	100	100	0	0	46.9	0	0	0	0	0	100	100	0	0	28.5	13.3	0
	Tabu search alone	0	0	0	100	100	0	0	0	100	100	0	0	0	100	100	0	0	0	100	100
	Without PC	0	0	0	100	100	0	0	0	100	100	0	0	0	100	100	0	0	0	100	100
	LSO-NIO	0	0	5.6	100	0	6	7	22	33.3	0	0	0	30	33.3	0	7.7	5	32.2	13.3	0
Hanoi	AMLSI	0	0	0.9	100	100	0	0	0.9	100	100	0	0	0.9	100	100	0.5	0.9	2	93	86.7
	Generation step	0	0	0.9	100	100	0	0	37.8	4.3	0	0	0	23.6	0	0	0	0	38	4.3	0
	Simple refinement	0	0	0.9	100	100	0	0	34.8	5	0	0	0	0.9	100	100	3.4	0	35	0.7	0
	Tabu search alone	0.8	0	12.7	76.7	0	0	0	12.8	71.33	0	3.7	6.6	18.8	53.3	6.7	4.3	6	19.7	49.3	6.7
	Without PC	15.6	30.1	21.8	8.3	6.7	0	7.1	14.5	68.7	13.3	15.6	30.1	21.7	8.3	6.7	3.4	7	14	68	26.7
	LSO-NIO	0	0	9.2	100	0	6.2	6.5	21.3	19.7	0	0	0	29.9	12.7	0	1.8	5	30.9	6	0
N-Puzzle	AMLSI	0	0	5.6	100	100	0	0	5.6	100	100	0	0	5.6	100	100	0	0	5.6	100	100
	Generation step	0	0	5.6	100	100	0	0	37.4	13.3	0	0	0	27	1.3	0	0	0	38.9	0	0
	Simple refinement	0	0	5.6	100	100	0	0	36	26.7	1.33	0	0	5.6	100	100	0	0	38.9	0	0
	Tabu search alone	0	0	5.6	100	100	0	0	5.6	100	100	0	0	5.6	100	100	0	0	5.6	100	100
	Without PC	1.1	1.7	6.7	93.3	93.3	0	0	5.6	100	100	1.1	1.7	6.7	93.3	93.3	0	0	5.6	100	100
	LSO-NIO	0	0	5.6	–	-	0	0	10.7	–	-	0	0	28.1	–	-	0	0	27.7	–	-
Peg Solitaire	AMLSI	0	0	6.7	100	100	3	0	12.3	66.7	66.7	0	0	7.3	100	100	4	0.8	14	80	76.7
	Generation step	0	0	4	0	0	0	0	30.2	0	0	0	0	21.2	0	0	0	0	30.1	0	0
	Simple refinement	0	0	4.2	0	0	0	1	30.2	0	0	0	0	4.2	100	0	0	1	30.1	0	0
	Tabu search alone	0	0	8.2	0	0	0	0	8.1	0	0	0	0	8.4	20	20	0	0	8.3	20	20
	Without PC	14	28	18.2	–	-	4.2	9.2	15.7	–	-	8.8	18.8	18.1	–	-	7.1	8.6	16.9	–	-
	LSO-NIO	0	0	3.6	–	-	2.9	11.5	15.3	–	-	0	0	20.8	–	-	8.4	7.1	24.6	–	-
Parking	AMLSI	0	0	3.9	–	-	0	0	3.9	–	-	0	0	4.1	–	-	0.2	0	4.3	–	-
	Generation step	0	0	3.9	–	-	0	0	30.2	–	-	0	0	21.2	–	-	0	0	30.9	–	-
	Simple refinement	0	1	4.2	–	-	0	0	30.1	–	-	0	1	4.2	–	-	0	0	30.7	–	-
	Tabu search alone	0	0	8.2	–	-	0	0	8.1	–	-	0	0	8.4	–	-	0	0	8.3	–	-
	Without PC	–	–	–	–	-	–	–	–	–	-	–	–	–	–	-	–	–	–	–	-
	LSO-NIO	–	–	–	–	-	–	–	–	–	-	–	–	–	–	-	–	–	–	–	-
Zenotravel	AMLSI	0	0	0	100	100	0.1	0	0.2	100	99.3	0	0	0	100	100	0	0	0	100	100
	Generation step	0	0	0	100	100	0	0	27.4	14.7	0	0	0	10	12.6	0	1.7	3	25.9	19.3	0
	Simple refinement	0	0	0	100	100	0.3	1.9	22.4	21.3	0	0	0	1	92.7	65.6	2.5	2.4	206	51.3	0
	Tabu search alone	0	0	1.8	100	73.3	0	0	5.4	100	33.3	0	0	4.2	100	40	0	0	8.8	100	20
	Without PC	5	14.5	9.6	–	-	2.1	9.4	8.4	–	-	4.9	13.5	9.4	–	-	2.8	8.9	7.5	–	-
	LSO-NIO	0	0	9.4	–	-	10.1	10	21.3	–	-	0	0	25.4	–	-	10.2	7.9	28.4	–	-
Sokoban	AMLSI	0	0	3.9	100	100	0	0	3.9	100	100	0	0	3.9	100	100	0	0	5.7	99.7	66.7
	Generation step	0	0	3.9	100	100	0	0	19.9	20	0	0	0	15	0	0	0	0	19.8	6.7	0
	Simple refinement	0	0	3.9	100	100	0.2	0	18.9	13.3	0	0	0	6.6	6.7	6.7	0	0	18.6	6.7	0
	Tabu search alone	0	0	3.9	100	100	0	0	3.9	100	100	0	0	3.9	100	100	0	0	4.3	99.7	93.3
	Without PC	4.4	7	13.9	–	-	0	0	3.9	–	-	4.2	6.6	13.5	–	-	0	0.1	6.2	–	-
	LSO-NIO	–	–	–	–	-	–	–	–	–	-	–	–	–	–	-	–	–	–	–	-
Neg Visit All	AMLSI	0	0	0	100	100	0	0	0	100	100	0	0	0	100	100	0	0	0	100	100
	Generation step	0	0	0	100	100	0	0	13.1	60	0	0	0	4.8	20	1	0	3.4	13.5	46.7	0
	Simple refinement	2.5	16.0	3.3	40	40	0	0	11.3	93.3	20	3	28.4	6.7	26.7	1.3	1.4	8.6	12.8	66.7	6.7
	Tabu search alone	0	0	0	100	100	0	0	0	100	100	0	0	0	100	100	0	0	0	100	100
	Without PC	7.3	18.3	18.1	46.7	26.7	1.7	3.1	7.2	86.7	80	8	16.1	17.6	26.7	26.7	1.6	1.9	4.4	86.7	86.7
	LSO-NIO	0	0	11.1	100	0	21.9	10.8	17.6	93.3	0	0	0	15.2	100	0	21.4	14.5	19.3	73.3	0
Neg Elevator	AMLSI	0	0	0	100	100	0	0	0	100	100	0	0	0	100	100	0	0	0	100	100
	Generation step	0	0	0	100	100	0	7.1	19	13.3	0	0	0	7.4	6.7	0	0	0	17.2	33.3	0
	Simple refinement	0	0	0	100	100	1.6	0.6	14	6.7	0	0	0	2	46.7	46.7	0.2	0	14.2	33.3	0
	Tabu search alone	0	0	5.3	100	66.7	0	0	5.3	100	66.7	0	0	5.3	100	66.7	0	0	3.2	100	80
	Without PC	0.5	0.8	2.4	66.7	66.7	0	0	0.1	100	100	0.4	0.5	1.8	73.3	73.3	0	0	0.9	93.3	93.3
	LSO-NIO	0	0	10.8	100	0	4.8	4.6	16.9	60	0	0	0	19.5	53.3	0	6.8	7.4	21.7	20	0

Table 3: Domain learning results on 10 IPC domains. AMLSI performance is measured in terms of error rates for preconditions (resp. effects) E_ρ (resp. E_ϵ), syntactical error E_σ, the rate of solved problem $Solved$ and accuracy Acc. AMLSI's performance are compare with LSO-NIO's performance with the same experimental setup. We report the averages computed over 15 runs (5 runs over 3 different initial states).

Impact of noisy and partial observations The results of the first scenario (complete intermediate observations and no noise) show that AMLSI perfectly learns the preconditions and the effects of the operators of the IPC domains. Note that for 4 IPC domains, Hanoi, N-Puzzle, Peg-Solitaire and Parking, the syntactical error is not equal to 0. This is because AMLSI learns preconditions that are not in the IPC domain. On the other hand, the obtained accuracy is optimal for all domains. This means that the domains learned with AMLSI can be used to solve all the problems of the IPC

Domain	$\#States$	$\#Nodes$	$\#Transitions$	Compression level
Blocksworld	449.7	23	43.4	19.7
Gripper	455	8	16	58.9
Hanoi	443.6	27.9	51.47	16.7
N-Puzzle	455.4	12.7	24.7	35.9
Peg-Solitaire	207.1	34.3	45.9	6.6
Parking	450.4	78.7	175.3	6.1
Zenotravel	453.1	24.1	53	19.5
Sokoban	450.7	31.3	64.1	16.8
Neg Visit All	449.7	39.7	64.5	12.4
Neg Elevator	451.5	24.5	44.4	18.9

Table 4: Induced automaton characteristics (from left to right): average number of states in the learning set, average number of nodes, average number of transitions, compression level, i.e. average number of states per node.

domains.

The results of the second scenario (complete intermediate observations and high level of noise (20%) are almost similar to the first scenario. Noise slightly reduces the quality of learning in terms of syntactical error and accuracy for two IPC domains (Blocksworld and Zenotravel). For Peg Solitaire, the impact is stronger, however the majority of the problems are correctly solved ($Acc = 63.3\%$). The impact of noise on the performance of AMSLI with complete intermediate observations is generally low.

The results of the third scenario (partial intermediate observations (25%) and no noise) are almost similar to the first scenario. Partial observation slightly reduces the quality of learning in terms of syntactical error and accuracy for one IPC domain (Parking). The impact of partial observation on the performance of AMLSI with noiseless observations is low.

Finally, the results of the fourth scenario (partial intermediate observations (25%) and high level of noise (20%)) show that partial intermediate observations downgrade the global performances of AMLSI. However, they remains high for all the IPC domains and metrics, and specifically for the accuracy. Despite partial and noisy observations, AMLSI is able to generate accurate PDDL domains. To our best knowledge, AMLSI is the only approach able to generate usable planning domains with these levels of noisy and partial observations.

Moreover, we can observe that domains including both positive and negative preconditions (Neg-Visit-All and Neg-Elevator) are easier to learn that domains including only positive preconditions. This is due to the fact that negative preconditions have a stronger impact on the fitness score of tabu search. Then, for domain using only positive preconditions, the easiest domains to learn (Gripper and N-Puzzle) are domains with the highest level of compression (see Table - 4) in the induced automaton, and the most difficult domain to learn (Peg Solitaire) is the domain with the lowest level of compression.

Impact of the initial state The choice of the initial state impacts the quality of the induced grammar induction used to generate the PDDL operators when domains. Table - 5 shows the standard deviation between the different initial states for each experimental scenario and for each of performance metrics. We can observe that for some domains the choice of the initial state has little to no impact on AMLSI's performances, e.g., Gripper, N-Puzzle, Neg-Visit-All, Neg-Elevator or Zenotravel. In other domains, e.g., Blocksworld, Hanoi, Parking and Peg Solitaire the impact is significant.

The performance gap between initial states can be explained by the different complexities of the grammars to induce. For instance, for the Peg Solitaire domain, the automaton learnt with the first initial state contains generally 20 nodes and 40 transitions and the compression level varies between 7 and 9, while the automaton learnt with the third initial state contains generally 50 nodes and 70 transitions and the compression level varies between 4 and 6. It is, therefore, easier to learn the domain with the first initial state than the third initial states. Also, the choice of the initial state impacts the coverage of the random walk in the generation step that can cause overfitting.

Ablation study In addition, we perform an ablation study of the AMLSI approach. We test each part of the AMLSI approach independently:

1. **Generation Step:** We learn domain by taking into account only the operator induction step. We can observe that for the majority of domains, this step is sufficient for the first experimental scenario (complete and noiseless observations). However when observations are partial and/or noisy, this step is not able to learn domains.
2. **Simple refinement:** We learn refined domain by taking into account only preconditions/effects precondition step, ie, without the Tabu search. As for the generation step, this step is generally able to learn domains when observations are complete and noiseless. In addition, this step is able to learn some domains (Blocksworld, Gripper, Hanoi and N-Puzzle) when observations are partial and noiseless. However when observations are noisy, this step is not able to learn domains. This is due to the fact that, during the mapping, noisy propositions will delete a large amount of information. The effect refinement steps will therefore no longer be able to detect all of the missing effects. In addition, when observations are noisy, there is a high risk that this step will detect additional effects that will not be removed by Tabu research.
3. **Tabu search alone:** We learn domain with only the Tabu search without the operator induction and preconditions/effects refinement steps. We can observe that for the majority of domains, tabu research alone is not capable of learning domains, whatever the experimental scenario. We can also note that taboo research alone generally learns the best areas when observations are noisy. Tabu search is therefore not sufficient to learn domains, and AMLSI is not able to learn domains without Tabu research either. This implies that it is the combination of tabu search effects and preconditions refinement step that allow AMLSI to learn accurate domains in each experimental scenario.

Grammar induction without pairwise constraints Then, we test a variant of the AMLSI algorithm where the grammar induction was performed without pairwise

Fluents	100%										25%									
Noise	0%					20%					0%					20%				
Domain	$E_\rho(\%)$	$E_\epsilon(\%)$	$E_\sigma(\%)$	$Solved(\%)$	$Acc(\%)$	$E_\rho(\%)$	$E_\epsilon(\%)$	$E_\sigma(\%)$	$Solved(\%)$	$Acc(\%)$	$E_\rho(\%)$	$E_\epsilon(\%)$	$E_\sigma(\%)$	$Solved(\%)$	$Acc(\%)$	$E_\rho(\%)$	$E_\epsilon(\%)$	$E_\sigma(\%)$	$Solved(\%)$	$Acc(\%)$
Blocksworld	0	0	0	0	0	0.7	0.6	0.4	9	9	0	0	0	0	0	0.6	1.2	0.5	7.4	7.4
Gripper	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Hanoi	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0.3	0.7	1	9.2	9.4
N-Puzzle	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Peg Solitaire	0	0	0	0	0	1.9	0	4.1	47.1	47.1	0	0	0	1.6	0	2.1	0.6	1.9	16.3	20.1
Parking	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
Zenotravel	0	0	0	0	0	0.2	0	0.2	0	0.9	0	0	0	0	0	0	0	0	0	0
Sokoban	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0.5	0.4	9.4
Neg Visit All	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Neg Elevator	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Table 5: Standard deviation between initial states for all the domains and metrics

constraints (noted *Without PC* in Table - 3). First of all, we can observe that the results are generally worse for each domain and for each experimental scenario.

The performance gap can be explained by the quality of induced grammar. Indeed, as we have seen in section 4, RPNI is optimal if and only if samples are "characteristic". The construction of a characteristic sample is not feasible a priori and we can't assume that the dataset generation produces characteristic samples. That implies that grammars induced without PC are generally more general that grammars induced with PC, and so there are generally more unfeasible paths, ie paths present in the grammar induced and not present in the ground truth grammar, in grammars induced without PC. These unfeasible paths causes noises in the effects refinement, ie, when grammars are induced without PC, extra effects are detected during the effects refinement step.

Finally, we can note that, when grammars are induced without PC, domains are generally better when observations are noisy. This due to the fact that when observations are noisy, effects refinement step detect less effects, so the tabu search has greater importance, and tabu search being independent of the induced grammar, induce grammars without PC has less impact when observations are noisy.

Comparison with LSO-NIO Then, we compare performance of AMLSI and LSO-NIO (Mourão *et al.* 2012) which is the approach with the closest input setting. Indeed, LSO-NIO takes as input random walks including action failures with partial and noisy observations. LSO-NIO has been tested with I_+ and I_-. Random walks including action failures are obtained by merging I_+ and I_-. Note that, in our experiment LSO-NIO's training set contains between 2000 and 5000 with a majority of failed actions, while LSO-NIO was tested in (Mourão *et al.* 2012) with training set containing 20000 actions with an equal mixture of successful and unsuccessful actions. Also, for our experimentation, there are fewer objects in initial states than for the experimentation of (Mourão *et al.* 2012). We use our own implementation of the LSO-NIO approach[4].

We can observe that AMLSI outperforms LSO-NIO. These results can be explained in several ways. First of all, we generally have a lot of negative information than positive information. This is beneficial for AMLSI because it makes it possible to have a good automaton, moreover it makes it possible to effectively lead tabu search. This is handicapping for LSO-NIO because the updated weights of the different classifiers need more positive information. In addition, LSO-NIO learns effects and preconditions by taking actions one by one into. While AMLSI, during the Tabu search, learns effects and preconditions by taking into account all action sequences.

Finally, we can note that the results of LSO-NIO are close to the results of AMLSI with Simple refinement.

[4]Our implementation is available at https://www.dropbox.com/sh/3gpcdasycuy4r9h/AACYDrwNeseOMFLX6tlmMZcWa?dl=0

6 Conclusion

In this paper we present AMLSI, a novel algorithm to learn PDDL domains. We assume that it is possible to query a system to model and to collect partial and noisy observations. AMLSI is composed of four steps. The first step consists in building two training sets of feasible and infeasible action sequences. In the second step, AMLSI induces a regular grammar. The third step is the generation of the PDDL operators, and the last step refines the generated operators. Our experimental results show that AMLSI successfully learns PDDL domains with high levels of noise and incomplete observations.

The performance of AMLSI depends a lot on the induced regular grammar. If the grammar is too complex, or too simple, AMLSI becomes more sensitive to noise and partial observations. As the complexity of the grammar depends on the initial states, future works will focus on the selection of the initial states. Moreover, it should be possible to bias the training set generation in order to obtain the best possible grammar while minimizing the queries. Finally, AMLSI will be extended in order to learn more expressive PDDL operators with disjunctive preconditions, conditional effects and numerical functions.

Acknowledgements

This work was funded by the Circular project - IDEX.

References

Stephen Cresswell and Peter Gregory. Generalised domain model acquisition from action traces. In *ICAPS*, 2011.

Stephen Cresswell, Thomas Leo McCluskey, and Margaret Mary West. Acquisition of object-centred domain models from planning examples. 2009.

Richard Fikes and Nils J. Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. *Artif. Intell.*, 2(3/4):189–208, 1971.

Yolanda Gil. Learning by experimentation: Incremental refinement of incomplete planning domains. In *ICML 1994*, pages 87–95, 1994.

Peter Gregory and Stephen Cresswell. Domain model acquisition in the presence of static relations in the LOP system. In *ICAPS*, pages 97–105, 2015.

Peter Gregory and Alan Lindsay. Domain model acquisition in domains with action costs. In *ICAPS*, pages 149–157, 2016.

Malte Helmert. The Fast Downward planning system. *Artif. Intell.*, 26:191–246, 2006.

Richard Howey and Derek Long. Val's progress: The automatic validation tool for pddl2. 1 used in the international planning competition. In *Proc. of ICAPS Workshop on the IPC*, pages 28–37, 2003.

Jirí Kucera and Roman Barták. LOUGA: learning planning operators using genetic algorithms. In *PKAW 2018*, pages 124–138, 2018.

Drew McDermott, Malik Ghallab, Adele Howe, Craig Knoblock, Ashwin Ram, Manuela Veloso, Daniel Weld, and David Wilkins. PDDL-the planning domain definition language, 1998.

Kira Mourão, Luke S. Zettlemoyer, Ronald P. A. Petrick, and Mark Steedman. Learning STRIPS operators from noisy and incomplete observations. In *UAI 2012*, pages 614–623, 2012.

Jose Oncina and Pedro García. Inferring regular languages in polynomial update time. In *Pattern Recognition and Image Analysis: Selected Papers from the IVth Spanish Symposium*, volume 1, pages 49–61. World Scientific, 1992.

Christophe Rodrigues, Pierre Gérard, and Céline Rouveirol. Incremental learning of relational action models in noisy environments. In *ILP 2010*, pages 206–213, 2010.

José Á Segura-Muros, Raúl Pérez, and Juan Fernández-Olivares. Learning numerical action models from noisy and partially observable states by means of inductive rule learning techniques. In *KEPS 2018*, pages 46–53, 2018.

Dafna Shahaf and Eyal Amir. Learning partially observable action schemas. In *AAAI 2006*, pages 913–919, 2006.

Xuemei Wang. Learning by observation and practice: An incremental approach for planning operator acquisition. In *ICML 1995*, pages 549–557, 1995.

Qiang Yang, Kangheng Wu, and Yunfei Jiang. Learning action models from plan examples using weighted MAX-SAT. *Artif. Intell.*, 171(2-3):107–143, 2007.

Hankz Hankui Zhuo, Qiang Yang, Derek Hao Hu, and Lei Li. Learning complex action models with quantifiers and logical implications. *Artif. Intell.*, 174(18):1540–1569, 2010.

Hankz Hankui Zhuo, Tuan Anh Nguyen, and Subbarao Kambhampati. Refining incomplete planning domain models through plan traces. In *IJCAI 2013*, pages 2451–2458, 2013.