IncrAMLSI: Incremental Learning of Accurate Planning Domains from Partial and Noisy Observations

Maxence Grand, Humbert Fiorino, Damien Pellier

IncrAMLSI: Incremental Learning of Accurate Planning Domains from Partial and Noisy Observations

1st Maxence Grand
Univ. Grenoble Alpes, LIG
38000 Grenoble, France
Maxence.Grand@univ-grenoble-alpes.fr

2nd Humbert Fiorino
Univ. Grenoble Alpes, LIG
38000 Grenoble, France
Humbert.Fiorino@imag.fr

3rd Damien Pellier
Univ. Grenoble Alpes, LIG
38000 Grenoble, France
Damien.Pellier@imag.fr

***Abstract*—Hand-encoding PDDL domains is generally considered difficult, tedious and error-prone. Many different machine learning techniques have been proposed to deal with these issues. Recent novel approaches such as AMLSI (*Action Model Learning with System Interaction*) achieve high level of accuracy, i.e. the learnt domains are accurate enough to solve planning problems without human proofreading. However, in most of the real world applications, training datasets are difficult and costly to acquire and become available gradually over time. To tackle this issue, we present IncrAMLSI, which is a version of AMLSI for incremental training datasets. We show that IncrAMLSI outperforms AMLSI to learn IPC benchmarks and converges with few learning iterations from partial and noisy data.**

I. Introduction

Hand-encoding PDDL domains is generally considered difficult, tedious and error-prone. The reason is that the experts of the domains to model are not always PDDL experts and vice versa. To overcome this issue, two main approaches have been proposed. One is to develop knowledge engineering tools facilitating PDDL writing, e.g., GIPO [1], EUROPA [2], itSimple [3], PDDL Studio [4]. These tools provide support for consistency and syntactic error checking, domain visualisation etc. An inconvenient of these tools is that they require PDDL expertise, or common knowledge in software engineering [5].

Another approach consists in developing machine learning algorithms to automatically produce PDDL domains, and avoid PDDL expertise: for instance, ARMS [6], SLAF [7], Louga [8], LSONIO [9], LOCM [10], IRale [11], PlanMilner [12]. In this approach, training data are (possibly noisy and partial) intermediate states, and plans previously generated by a planner, or randomly generated action sequences. The challenge is then to rebuild the PDDL domains that were used to generate these states and plans.

Learning techniques for PDDL domains are promising but they are still in the preliminary stages. One major drawback of these techniques is the *accuracy* of the generated domains, which measures the capacity to solve planning problems that were not in the training datasets. Thus these domains are not directly usable by planners, and a stage of domain proofreading by an expert is almost always necessary to correct errors or inaccuracies in the predicates of the actions.

Recently, [13] have proposed a novel approach called AMLSI (*Action Model Learning with System Interaction*) to address the accuracy problem. AMLSI is based on grammar induction [14]. It takes as input noisy and partial state observations and learns planning domains accurate enough to be used by planners to solve new planning problems despite high levels of noise in observations. A specific feature of AMLSI is that it works with both feasible and infeasible action sequences generated from random walks while most approaches use only feasible action sequences or plan traces as training datasets.

However, another important drawback of the learning techniques including AMLSI is that, in pratice, dataset acquisition is a long term evolutive process: in real world applications, training data become available gradually over time, are difficult and costly to obtain, as for instance, Mars Exploration Rover operations [15] or robot fleets for offshore missions [16]. Moreover, in practice, it is important to be able to update learnt planning domains to new incoming data without restarting the learning process from scratch.

In this paper, we propose an incremental version of AMLSI called IncrAMLSI. To our best knowledge, IncrAMLSI is the only approach in the literature able to incrementally learn accurate planning domains with noisy and partial observations: other approaches learn in batch mode from a given set of states/plans, while IncrAMSLI learns incrementally from incoming data, and does not restart from scratch when new data become available.

The rest of the paper is organized as follows. In section II we propose a problem statement. In section III we give some backgrounds on the AMLSI approach and, in section IV, we detail the incremental extension of AMLSI. Then, the section V evaluates the performance of IncrAMLSI on IPC benchmarks. We show that IncrAMLSI outperforms AMLSI in terms of accuracy when training data are noisy and partially observable, and converges with very few iterations. Finally, in section VI we present the related works.

II. Formal Framework

A STRIPS planning problem is a tuple $P = (L, A, S, s_0, G, \delta, \lambda)$, where L is a set of propositions using to

describe the world, S is a set of state labels, $s_0 \in S$ is the label of the initial state and $G \subseteq S$ is a set of goal state labels. Note we will often just refer to its elements as state, though these are just the labels of the states. λ is an observation function $\lambda : S \to 2^L$ that assigns to each state label the set of atomic propositions true in that state. A is a set of action labels or action names. In a similar way to the states, we will often just refer to its elements as actions, though these are just the labels of the actions that do not have the often-used tuple form. Their preconditions and positive as well as negative effects are given by the functions $prec$, add and del that are included in δ, i.e. $\delta = (prec, add, del)$. $prec$ is defined as $prec : A \to 2^L$. The functions add and del are defined in the same way. Without loss of generality, we chose this unusual formal framework inspired by [17] for defining S and A in order to facilitate the statement of the STRIPS learning problem.

The function $\tau : A \times S \to \{true, false\}$ returns whether an action is applicable to a state, i.e. $\tau(a, s) \Leftrightarrow prec(a) \subseteq \lambda(s)$. Whenever an action is applicable, the state transition function $\gamma : A \times S \to S$ returns the state resulting from applying an action to a state: $\gamma(s_i, a) = s_{i+1}$ such that $\lambda(s_{i+1}) = [\lambda(s_i) \setminus del(a)] \cup add(a)$.

A sequence $(a_0 a_1 \dots a_n)$ of actions is applicable to a state s_0 when each action a_i with $0 \leq i \leq n$ is applicable to the state s_i. Given an applicable sequence $(a_0 a_1 \dots a_n)$ in state s_0, $\gamma(s_0, (a_0 a_1 \dots a_n)) = \gamma(\gamma(s_0, a_0), (a_1 \dots a_n)) = s_{n+1}$. It is important to note that this recursive definition of γ entails to the implicit generation of a sequence of states $(s_0 s_1 \dots s_{n+1})$. A goal state is a state s such that $\lambda(g) \subseteq \lambda(s)$ with $g \in G$. $s \models g$, i.e. s satisfies g if and only if s is a goal state. An action sequence is a plan (also solution) to a planning problem P if and only if it is applicable to s_0 and results in a goal state when applied to s_0.

In formal languages, a set of rules is given that describe the structure of valid words and the language is the set of these words. For STRIPS planning problem $P = (L, A, S, s_0, g, \delta, \lambda)$, this language is defined as $(0 \leq i \leq n)$:

$$\mathcal{L}(P) = \{\omega = (a_0 a_1 \dots a_n) | a_i \in A, \gamma(s_0, \omega) \models g\} \quad (1)$$

We know that the set of languages generated by STRIPS planning problems are regular languages [17]. In other words, a STRIPS planning problem $P = (L, A, S, s_0, G, \delta, \lambda)$ generate a language $\mathcal{L}(P)$ that is equivalent to a Deterministic Finite Automaton (DFA) $\Sigma = (S, A, \gamma)$. S and A are respectively the nodes and the arcs of the DFA and γ the transition function.

For any arc $a \in A$, we call *pre-set* of a the set $\mu_{Ante} = \{s \in S \mid \gamma(s, a) = s'\}$ and *post-set* of a the set $\mu_{Post} = \{s' \in S \mid \gamma(s, a) = s'\}$. $\mu_{Ante}(s, a)$ (resp. $\mu_{Post}(s', a)$) gives the intersection of the observed state set before (resp. after) the execution of a in state s: a is an outcoming (resp. incoming) edge of s (resp. s') in the DFA.

A STRIPS learning problem is as follow: given a set of observations $\Omega \subseteq \mathcal{L}(P)$, is it possible to learn Σ and deduce P?

For instance, suppose $\Omega = \{a, ab, ba, bab, abb, \dots\}$ such that $s_0 \xrightarrow{a} s_2$, $s_0 \xrightarrow{a} s_2 \xrightarrow{b} s_2$, $s_0 \xrightarrow{b} s_1 \xrightarrow{a} s_2$, $s_0 \xrightarrow{b} s_1 \xrightarrow{a} s_2 \xrightarrow{b} s_2 \xrightarrow{b} s_2 \dots$ Can we learn Σ (see Figure 1) with actions $\{a, b\}$, the initial state s_0 and some states mark as goal $G = \{s_2\}$ so that all observations in Ω are a path from the initial state to a goal state to deduce P?

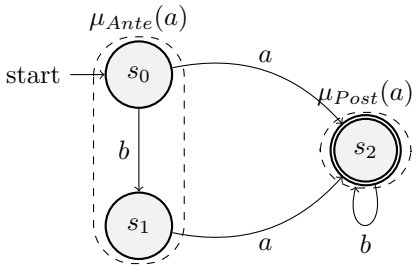


Fig. 1: An example of DFA with pre-set μ_{Ante} and post-set μ_{Post}

III. Background on AMLSI

In practice, AMLSI generates Ω as input using random walk to learn $\Sigma = (S, A, \gamma)$ and deduce $P = (L, A, S, s_0, G, \delta, \lambda)$. AMLSI assumes L, A, S, s_0 known and the observation function λ possibly partial and noisy (a partial observation is a state where some propositions are missing and a noisy obsevation is a state where the truth value of a proprosition is erroneous). No knowledge of the goal states G is required. Once Σ is learnt, it remains for AMLSI to deduce δ from the transition function γ. Concretely, δ can be represented as PDDL planning domain containing all the actions of the problem P and by induction the classical PDDL operators.

The AMLSI algorithm consists of 4 steps: (1) generation of the observations, (2) learning the DFA corresponding to the observations, (3) induces δ the PDDL operator from the learnt DFA; (3) finally, refining these operators to deal with noisy and partial state observations. The first step consisting to generate Ω. AMLSI uses random walk by applying an action from the initial state of the problem. If the action is applicable in the current state the sequence of actions from the initial state is valid and is added to I^+ (the set of positive samples) otherwise the random walk is stopped and the sequence is added to I_- (the set negative samples).

The second step consisting in learning the DFA $\Sigma = (S, A, \gamma)$ is carried out by using a variant of the classic algorithm for learning regular grammar called RPNI [18] proposed by [19] based on I^+ and I^-.

The third step consists in generating the PDDL operators of the planning domain to learn. AMLSI begins by inducing the preconditions and effects of actions. To learn the preconditions $prec$ of an action a, AMLSI find all propositions that are always present in the pre-set $\mu_{Ante}(s, a)$ of all the state where a is feasible. Formally, p $(resp\ \neg p) \in prec(a)$ if and only if:

$$\forall (s, a) \in \Sigma : p\ (resp\ \neg p) \in \mu_{Ante}(s, a) \quad (2)$$

Then, To learn the positive (resp. negative) effects add (resp. del) of an action a, AMLSI finds all propositions that are

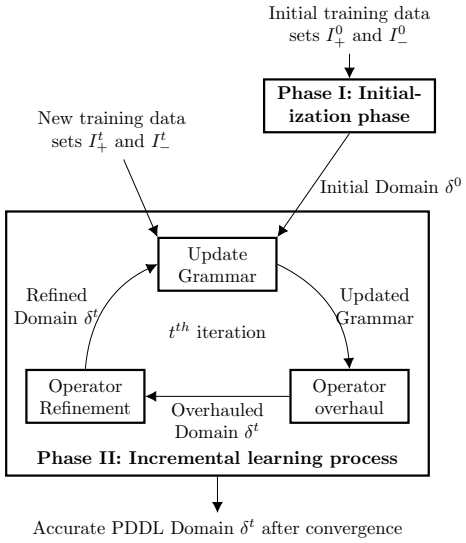


Fig. 2: IncrAMLSI overview algorithm

always (resp never) present before the execution of a in the DFA and never (resp always) present after the execution. Formally, $p \in add(a)$ if and only if:

$$\forall (s, a, s') \in \Sigma : p \notin \mu_{Ante}(s, a) \wedge \neg p \in \mu_{Post}(s', a) \quad (3)$$

And $\neg p \in del(a)$ if and only if

$$\forall (s, a, s') \in \Sigma : p \in \mu_{Ante}(s, a) \wedge p \notin \mu_{Post}(s', a) \quad (4)$$

Once actions preconditions and effects have been learnt, ALMSI generalize actions as PDDL operators and obtain a PDDL domain δ.

Finally, the last step consists in refining the PDDL operators induced at step 3 to deal with noisy and partial state observations. First of all, AMLSI starts by refining the operator effects to ensure that the generated operators allow to regenerate the induced DFA. To that end, AMLSI adds all effects allowing to ensure that each transition in the DFA are feasible. Then, AMLSI refines the preconditions of the operators. AMLSI makes the following assumptions as in [6]: the negative effects of an operator must be used in its preconditions. Thus, for each negative effect of an operator, AMLSI adds the corresponding proposition in the preconditions. Since effect refinements depend on preconditions and precondition refinements depend on effects, AMLSI repeats these two refinements steps until convergence, i.e., no more precondition or effect is added. Finally, AMLSI performs a Tabu Search to improve the PDDL operators independently of the induced DFA, on which operator generation is based. Once the Tabu Search reaches a local optimum, AMLSI repeats all the three refinement steps until convergence.

IV. Incremental AMLSI

An overview of incremental AMLSI (IncrAMLSI) is shown in Figure 2. It learns incrementally from incoming data and does not restart from scratch when new data become available at each iteration t. It consists of two phases:

Algorithm 1: Overhaul_Operator(δ^t, I_+^t, I_-^t)

```
1  backtrackInfeasibleEffects(δ^t);
2  backtrackInfeasiblePreconditions(δ^t);
3  repeat
4      repeat
5          backtrackPreconditions(δ^t);
6          backtrackNegativeEffects(δ^t);
7      until acceptAll(δ^t, I_+^t);
8      backtrackEffects(δ^t)
9  until acceptAll(δ^t, I_+^t);
10 δ^new ← operatorGeneration();
11 δ^new ← DFARefinement(δ^new);
12 δ^{t+1} ← merge(δ^t, δ^new);
13 return δ^{t+1}
```

- *Phase 1. (Initialising phase)* consists in generating the initial PDDL domain δ^0 from the initial training datasets I_+^0 and I_-^0 by applying the AMLSI algorithm as described in section III,
- *Phase 2. (Incremental learning phase)* consists in incrementally updating the PDDL domains δ^t with the new incoming training datasets I_+^t and I_-^t available at iteration t to produce the domain δ^{t+1}. This phase is made up of three steps:
  1) *Update of the DFA (regular grammar)* with the modified version of RPNI algorithm described in [19] in order to accept I_+^t and to reject I_-^t at iteration t;
  2) *Overhaul of the PDDL operators* in order to remove preconditions and effects that are no longer compatible with I_+^t and I_-^t, and the updated DFA at iteration t,
  3) *Refinement of the PDDL operators* as in AMLSI to deal with noisy and partial states in I_+^t and I_-^t, and to produce the new domain δ^{t+1}.

The incremental learning phase is operated each time new training datasets are input and until convergence of the PDDL domain. In the rest of this section, we detail the overhaul step (sub-section IV-A) and the convergence criterion (sub-section IV-B). The steps 1 and 3 are identical to AMLSI.

A. Operator overhaul

When at iteration t new positive and negative samples are integrated and the DFA is updated, it is possible that the domain previously learnt δ^t is no longer compatible with it. There are two possibilities: either some effects and preconditions in δ^t operators have to be removed to match the updated DFA, or it requires to add some preconditions and effects in the operators of δ^t. Algorithm 1 describes the procedure to compute δ^{t+1} from δ^t, I_+^t, and I_-^t at iteration t. First of all, functions *backtrackInfeasibleEffects* and *backtrackInfeasiblePreconditions* (line 1, 2) remove the effects and preconditions infeasible in the DFA. More precisely, *backtrackInfeasibleEffects* removes positive and nega-

tive effects that does not respect the conditions given by the equations 3 and 4 in section III. For instance, suppose we have $(holding\ ?x_2) \in add(unstack(?x_1\ ?x_2))$, and there are no states in the current DFA where this effect appears, then $(holding\ ?x_2)$ is removed. Likewise, *backtrackInfeasiblePreconditions* removes extra preconditions that does not validate equation 2. For instance, suppose we have $(holding\ ?x_2) \in prec(unstack(?x_1\ ?x_2))$, if the action $unstack(a, b)$ is feasible in state s, and $(holding\ b) \notin \mu_{Ante}(s, stack(a\ b))$, then $(holding\ ?x_2)$ is removed.

Then, as observed states are noisy and partial, it is possible that some extra preconditions and effects are not present in the current DFA. Therefore lines 3 - 9 remove extra preconditions and effects independently of this DFA. First of all, *backtrackPreconditions* (line 5) removes all the preconditions that are not compatible with I^t_+. For instance, suppose we have this positive sample:

$$[pick\_up(a), put\_down(a), pick\_up(a), \\ stack(a, b), unstack(a, b)] \in I^t_+$$

and, when we execute it with the current domain δ^t, $(holding\ a) \notin \lambda(\gamma(s_0, [pick\_up(a), put\_down(a), pick\_up(a), stack(a, b)]))$, i.e. in the observed state before the last action $unstack(a, b)$, and $(holding\ ?x_1) \in prec(unstack(?x_1\ ?x_2))$. Thus we remove this unsatisfied precondition $(holding\ ?x_1)$ from $unstack(?x_1\ ?x_2)$.

Then, with *backtrackNegativeEffects* (line 6), we remove all the negative effects that does not satisfy PDDL syntactical constraints [6] (e.g. negative effects must be in the preconditions etc.) We repeat these two functions until the domain accepts all the positive samples (line 7), i.e. are able to regenerate all the positive samples I^t_+.

The next step (line 8) in the algorithm is to remove all the effects that are not compatible with the negative samples I^t_- with function *backtrackEffects*. For instance, suppose that two samples share the same prefix $unstack(a, b)$:

$$[unstack(a, b), put\_down(a)] \in I^t_+$$

and

$$[unstack(a, b), put\_down(b)] \in I^t_-$$

Also, suppose that, wrongly, $(holding\ ?x_2) \in add(unstack(?x_1\ ?x_2))$, and, correctly, $(holding\ ?x) \in prec(put\_down(?x))$. And suppose that the current domain δ^t can generate $unstack(a, b)$. The wrong effect $(holding\ b) \in add(unstack(a, b))$ allows to the precondition of the unfeasible action $put\_down(b)$ to be satisfied, and $(holding\ b)$ is not in the preconditions of the feasible action $put\_down(a)$. Thus, $(holding\ ?x_2)$ must be removed from $add(unstack(?x_1, ?x_2))$.

As the *backtrackEffects* function removes effects, it is possible that some positive samples can no longer be generated by the current domain δ^t. We therefore have to repeat these three backtracking functions until the domain accepts all the positive samples (line 9). Indeed, repeating these functions until the domain accepts all the positive samples allows to ensure that the last removed effects do not impact the acceptation of the positive samples (termination is ensured by only allowing precondition and/or effect withdrawals).

Finally, the previous backtracking steps (line 3 - 9) allow to remove all the preconditions and effects that are not compatible with the DFA and the samples: all the wrong preconditions/effects have been removed, as well as possibly some correct ones. The objective of lines 10 - 11 is to integrate into δ^t all the remaining information in the DFA. This cannot corrupt δ^t because, by definition, the DFA accepts I^t_+ and rejects I^t_-. We build a new domain δ^{new} (line 10) by generating the operators as in Phase 1. Then, the effects and preconditions of δ^{new} are refined (line 11) as in AMLSI [13]. Finally, we merge δ^{new} with the current domain δ^t (line 12) by making the union sets of the preconditions and the effects of the operators.

B. Convergence criterion

IncrAMLSI stops when, after a given number Δ of iterations, the domain δ^t checks three conditions: (1) δ^t accepts all the positive samples I^t_+, (2) δ^t rejects all the negative samples I^t_-, and (3) $\delta^t = \delta^{t+1}$ during Δ iterations. We will show experimentally (see section V) that IncrAMLSI with partial and noisy state observations converges to a domain D. However, it is possible to show formally that it is true when state observations are complete and noiseless and $\Delta \to \infty$. The sketch of the proof is as follows: RPNI algorithm, which is at the core of IncrAMLSI, converges if the training dataset is a "characteristic sample" [18], i.e. the dataset covers all the states and transitions of the targeted grammar.

V. Experiments

In order to compare AMLSI with IncrAMLSI, we use the same experimental framework as in [13]. The experiments are based on 10 IPC domains: Blocksworld, Gripper, Hanoi, N-Puzzle, Peg Solitaire, Parking, 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 domains[1].

To deal with the fact that the initial state chosen may impact the learning process, we test each IPC domain with 3 different initial states over five runs and we used five seeds randomly generated for each run. Then, the length of the random walk sequences generated is randomly chosen between 10 and 20. Finally we generate partial observations by randomly removing a fraction of the propositions of the states and we generate noise changing the value of a fraction of observable proposition of the states. 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.

[1]The experimental setup and results including additional experimental results are available at https://gitlab.com/AMLSI/amlsi.

Observability		100%						25%					
Noise		0%			20%			0%			20%		
Domain	Algorithm	FScore	E_σ	Acc	FScore	E_σ	Acc	FScore	E_σ	Acc	FScore	E_σ	Acc
Blocksworld	AMLSI	100	0	100	94.6	0.6	93.7	100	0	100	77	1.2	76.3
	IncrAMLSI	1000	0	100	**100**	**0**	**100**	100	0	100	**100**	**0.4**	**94.3**
Gripper	AMLSI	100	0	100	100	0	100	100	0	100	100	0	100
	IncrAMLSI	100	0	100	100	0	100	100	0	100	100	0	100
Hanoi	AMLSI	100	0.9	100	100	0.9	100	100	0.9	100	91.6	2	86.7
	IncrAMLSI	100	0.9	100	100	0.9	100	100	0.9	100	**100**	**0.9**	**100**
N-Puzzle	AMLSI	100	5.6	100	100	5.6	100	100	5.6	100	100	5.6	100
	IncrAMLSI	100	5.6	100	100	5.6	100	100	5.6	100	100	5.6	100
Peg Solitaire	AMLSI	100	4.2	100	96	7.1	93.3	100	4.2	100	97.9	6.9	**96.3**
	IncrAMLSI	100	4.2	100	100	**5.2**	100	100	4.2	100	**99**	**6.3**	95
Zenotravel	AMLSI	100	0	100	95	0.2	99.3	100	0	100	100	0	100
	IncrAMLSI	100	0	100	**100**	**0**	**100**	100	0	100	100	0	100
Parking	AMLSI	100	3.9	100	100	3.9	100	100	4.1	100	93.5	**4.2**	100
	IncrAMLSI	100	3.9	100	100	3.9	100	100	3.9	100	100	4.3	100
Sokoban	AMLSI	100	3.9	100	100	3.9	100	100	3.9	100	67.1	6.1	60
	IncrAMLSI	100	3.9	100	100	3.9	100	100	3.9	100	**86.1**	**5.5**	**73**
NegVisitAll	AMLSI	100	0	100	100	0	100	100	0	100	100	0	100
	IncrAMLSI	100	0	100	100	0	100	100	0	100	100	0	100
NegElevator	AMLSI	100	0	100	100	**0**	100	100	0	100	100	**0**	**100**
	IncrAMLSI	100	0	100	100	0.2	100	100	0	100	100	0.1	93.3

TABLE I: Comparison between AMLSI and IncrAMLSI with $\Delta = 10$. Best results for each metrics are in bold.

Observability	100%		25%	
Noise	0%	20%	0%	20%
Domain	Number of iterations			
Blocksworld	11.5	16.9	13.2	32.5
Gripper	11	11.2	11.3	14.4
Hanoi	11	13	13.2	22.2
N-Puzzle	11	11	11	11
Peg Solitaire	11.7	26.7	20.5	45.6
Zenotravel	11	14.5	12.8	19.2
Parking	11	13.4	20.3	34.7
Sokoban	11	15	14.8	55.5
Neg Visit All	11.7	12.3	11.2	11.9
Neg Elevator	11.1	13.5	13.9	24.7

TABLE II: Average number of iterations to converge when $\Delta = 10$

A. Evaluation Metrics

We evaluate IncrAMLSI with three different metrics: the *syntactical error* [20] that computes the distance between the original domain and the domain learnt, the *accuracy* [21] that express the capability of a planner to use the domain learnt to solve new problems. Even though the syntactical error is the most used metric in the literature, we argue that the accuracy is the most important metric *in pratice* for planning because it measures to what extent a learnt domain is useful. Indeed, it often happens that one missing precondition or effect, which amounts to a small syntactical error, makes them unable to solve planning problems. Finally, the last metric is the *FScore* that express the capability of the learnt domain to regenerate the regular grammar.

Formally, the syntactical error $error(a)$ for an action a is defined as the number of extra or missing predicates in the preconditions ρ_a, the positive effects ϵ_a^+ and the negative effects ϵ_a^- divided by the total number of possible predicates. By extension, the syntactical error for a domain composed of a set of actions A is:

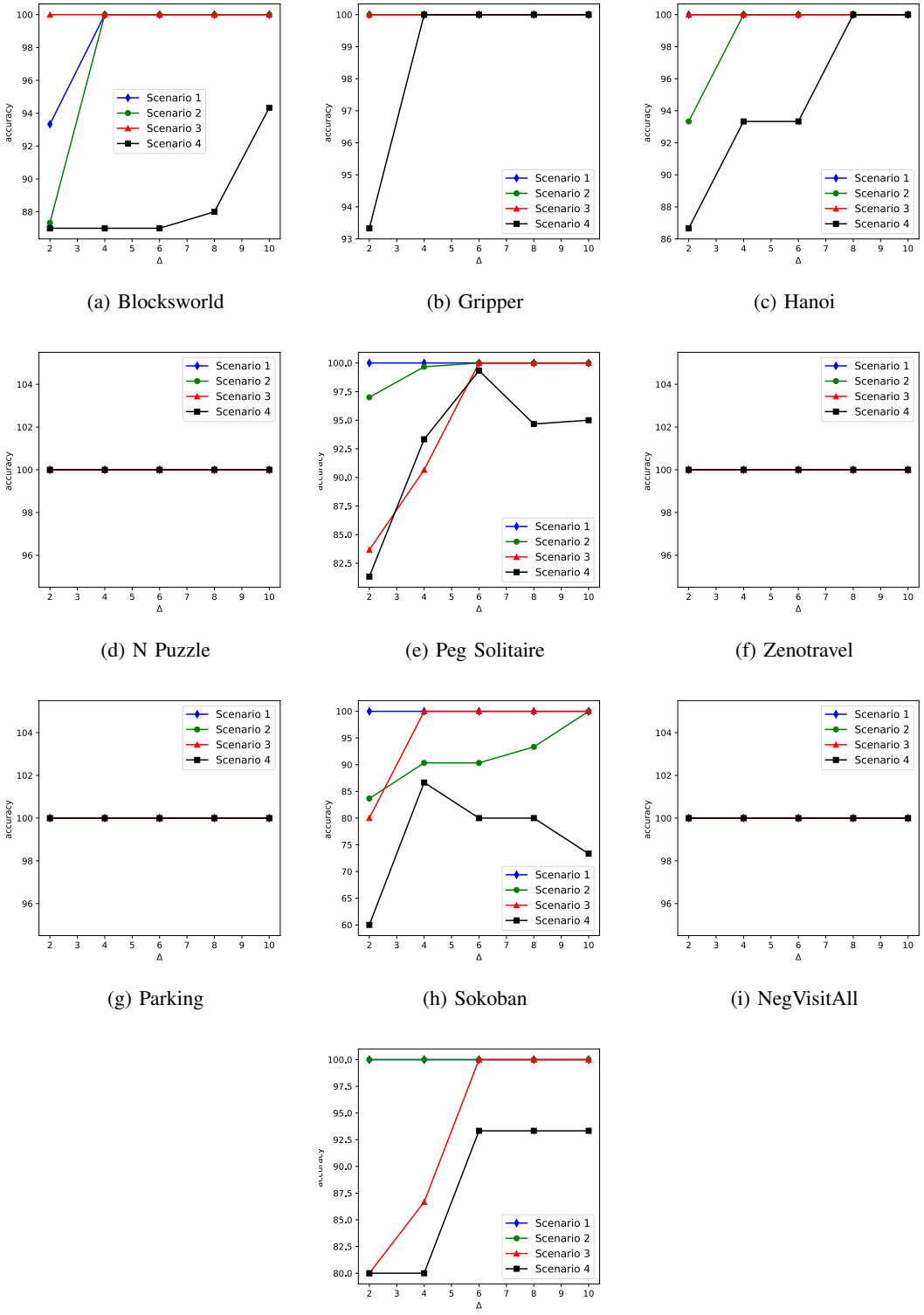
$$E_\sigma = \frac{1}{|A|} \sum_{a \in A} error(a)$$

Formally, the accuracy $Acc = \frac{N}{N^*}$ is the ratio between N, the number of correctly solved problems with the learnt domain, and N^*, the total number of problems to solve. In the rest of this section the accuracy is computed over 20 problems. The problems are solved with Fast Downward v19.06 [22]. Plan validation is realized with the automatic validation tool used in the IPC competition VAL [23].

Formally, the FScore is computed as follows: FScore $= \frac{2.P.R}{P+R}$ where R is the recall, i.e. the rate of sequences e accepted by the ground truth domain that are successfully accepted by the learnt domain, computed as follows: $R = \frac{|\{e \in E^+ \mid accept(D,e)\}|}{|E|}$ and P is the precision, i.e. the rate of sequences e accepted by the learnt domain that are sequences accepted by the ground truth domain, computed as follow: $P = \frac{|\{e \in E^+ \mid accept(D,e)\}|}{|\{e \in E^+ \mid accept(D,e)\} \cup \{e \in E^- \mid accept(D,e)\}|}$. The test set used to compute this metric is generated by a random walk using the observation generation method described in Section - III.

B. Results

In order to study the performance of IncrAMLSI with respect to noisy and partial state observations, we use four different experimental scenarios: (1) complete intermediate



(a) Blocksworld (b) Gripper (c) Hanoi

(d) N Puzzle (e) Peg Solitaire (f) Zenotravel

(g) Parking (h) Sokoban (i) NegVisitAll

(j) NegElevator

Fig. 3: Convergent learning while Δ varies

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%) and (4) partial intermediate observations (25%) and high level

of noise (20%).

First of all, Table I gives a comparison between AMLSI and IncrAMLSI. The results of the first scenario (complete intermediate observations and no noise) show that AMLSI and IncrAMLSI perfectly learn the preconditions and the effects of the operators of the IPC domains. For all algorithms and for all domains FScore and accuracy are optimal. Note that for 4 IPC domains, Hanoi, N-Puzzle, Parking and Sokoban, the syntactical error is not equal to 0. This is because algorithms learn implicit preconditions that are not encoded in the IPC domain. The results of the second scenario (complete intermediate observations and high level of noise (20%)) show that IncrAMLSI is generally more robust to noise than AMLSI when observations are complete. IncrAMLSI learns optimal domains for Blocksworld and Zenotravel and outperforms AMLSI. For Peg Solitaire, the IncrAMLSI algorithm outperforms AMLSI and obtains optimal accuracy, however the syntactical distance is deteriorated. The results of the third scenario (partial intermediate observations (25%) and no noise) are similar to the results of the first scenario. Finally, the results of the fourth scenario (partial intermediate observations (25%) and high level of noise (20%)) show that IncrAMLSI is generally the most robust algorithm. Indeed, in 3 IPC domains (Blocksworld, Hanoi, and Sokoban), IncrAMLSI learns better domains than AMLSI, while only 2 IPC domain (PegSOliatire and NegElevator) is better for AMLSI. To conclude, we observe that for all the domains, whatever the experimental scenario, IncrAMLSI always obtains a higher FScore than AMLSI. This is explained by the convergence criterion used by IncrAMLSI: the convergence criterion tends to learn a domain able to perfectly regenerate the grammar. However, we notice that a better FScore does not always guarantee a better accuracy. For instance, in the Peg Solitaire domain of the fourth experimental scenario, the FScore of IncrAMLSI is 99% while the FScore of AMLSI is 97.9%. But the accuracy of IncrAMLSI is 95% while the accuracy of AMLSI is 96.3%. However, we note that, in general, a better FScore is correlated with a better accuracy.

Then, Table II shows the average number of iterations for IncrAMLSI convergence. We note that for the first and the third scenario, IncrAMLSI converges in less than 20 iterations for each domain. For the second scenario we note that for each domain, except Peg Solitaire, IncrAMLSI converges in 15 iterations or less. Finally for the last scenario, IncrAMLSI needs more than 30 iterations for 4 IPC domains (Blocksworld, Peg Solitaire, Parking and Sokoban). However, for these domains, IncrAMLSI generally outperforms AMLSI.

Finally, Figure 3 shows the performance in terms of accuracy of the IncrAMLSI approach while Δ varies between 2 and 10. For each domain, increasing Δ makes it possible to improve the accuracy. We can notice that the Gripper, Hanoi, N-Puzzle, Zenotravel and NegVisitAll domains always converge towards the optimal domain whatever the scenario if Δ is high enough. For Blocksworld, when the observations are noisy and partial, the convergent algorithm seems converge if $\Delta > 10$. Finally, the domains Peg Solitaire, Sokoban and NegElevator seems to have trouble to converge, however learnt domains are always accurate ($Acc > 50\%$) whatever the level of noise and observability and the value of Δ for these domains.

VI. Related Works

Many approaches have been proposed to learn planning domains. We propose below a classification according to the input data of the learning process. The input data can be plan "traces" obtained by resolving a set of planning problems, partial planning domains to complete or "random walks". The input data can be complete (states and actions), partial, completely blind, or noisy.

A first group of approaches takes as input a set of plan traces and a partial domain, and tries to incrementally refine this domain to complete it, as for instance, OpMaker [24] or RIM [21]. In practice, RIM constructs sets of "soft" and "hard" constraints between observed states and actions, which are solved with weighted MAX-SAT solvers to generate refined actions. In all of these approaches, it is assumed that the observations are complete and noiseless. Opmaker works differently. It induces operators with user interactions. The partial domain is built by the user with the GIPO tools [1], then the user gives plan traces and intermediate state observations, and OpMaker induces pre-conditions and effects.

A second group of approaches takes as input only plan traces. Most of them deals with partial observations (except Observer [25] that needs complete observations). Among these approaches are ARMS [6], SLAF [7], or Louga [8]. ARMS gathers knowledge on the statistical distribution of frequent sets of actions in the plan traces. It then forms a weighted propositional satisfiability problem (weighted SAT) and solves it with a weighted MAX-SAT solver. Unlike ARMS, SLAF is able to learn actions with conditional effects. To that end, SLAF relies on building logical constraint formula based on a direct acyclic graph representation. Louga takes also as input plan traces and work with partial noiseless observations. However, Louga is able to learn actions with static properties and negative preconditions. Louga uses a genetic algorithm to learn action effects and an ad-hoc algorithm to learn action preconditions. Plan-Milner uses a classification algorithm based on inductive rule learning techniques: it learns action models with discrete numerical values from partial and noisy observations. Finally, the LOCM family of action model learning approaches [10], [26], [27], [28] works without information about initial, intermediate and final states. These algorithms extract, from plan traces, parameterized automata representing the behaviour of each object of the planning problems. Then preconditions and effects are generated from these automata.

The last group of approaches takes as input random walks, that is, sets of action sequences randomly generated. Random walk approaches like IRALe [11] deal with complete but noisy observations. IRALe is based on an online active algorithm to explore and to learn incrementally the action model with noisy observations. Others approaches such as LSO-NIO [9]

deal with both partial and noisy observations. LSO-NIO uses a classifier based on a kernel trick method to learn action models. It consists of two steps: (1) it learns a state transition function as a set of classifiers, and (2) it derives the action model from the parameters of the classifiers.

VII. Conclusion

In this paper we have addressed the problem of learning PDDL domains from plan traces with noisy and partial observations. To deal with this problem, we have presented an incremental learning approach called IncrAMLSI that achieves a high level of accuracy, i.e. the learnt domains are accurate enough to solve planning problems without human proofreading. A specific feature of IncrAMLSI is that it is able to deal with data that becomes available gradually over time. We have experimentally shown that IncrAMLSI (1) outperforms the most accurate approach existing in the literature, and (2) converges to the experimental domains with a few iterations from partial and noisy observations.

In future works, we will extend our approach to learn more expressive planning domains such as Temporal domains and HTN domains by building on the breakthroughs of grammar induction algorithms.

Acknowledgements

This research is supported by the French National Research Agency under the "Investissements d'avenir" program (ANR-15-IDEX-02) on behalf of the Cross Disciplinary Program CIRCULAR.

References

[1] R. M. Simpson, D. E. Kitchin, and T. L. McCluskey, "Planning domain definition using GIPO," *Knowledge Engineering Review*, vol. 22, pp. 117–134, 2007.

[2] J. Barreiro, M. Boyce, M. Do, J. Frank, M. Iatauro, T. Kichkaylo, P. Morris, J. Ong, E. Remolina, T. Smith *et al.*, "Europa: A platform for AI planning, scheduling, constraint programming, and optimization," in *In Proc. of the International Competition on Knowledge Engineering for Planning and Scheduling*, 2012.

[3] T. S. Vaquero, V. Romero, F. Tonidandel, and J. R. Silva, "itsimple 2.0: An integrated tool for designing planning domains," in *In Proc. of the International Conference on Automated Planning and Scheduling*, 2007, pp. 336–343.

[4] T. Plch, M. Chomut, C. Brom, and R. Barták, "Inspect, edit and debug pddl documents: Simply and efficiently with pddl studio," in *In Proc. of the ICAPS System Demonstrations and Exhibits*, 2012, pp. 15–18.

[5] M. Shah, L. Chrpa, F. Jimoh, D. Kitchin, T. McCluskey, S. Parkinson, and M. Vallati, "Knowledge engineering tools in planning: State-of-the-art and future challenges," *Knowledge engineering for planning and scheduling*, vol. 53, p. 53, 2013.

[6] Q. Yang, K. Wu, and Y. Jiang, "Learning action models from plan examples using weighted MAX-SAT," *Artificial Intelligence*, vol. 171, no. 2-3, pp. 107–143, 2007.

[7] D. Shahaf and E. Amir, "Learning partially observable action schemas," in *In Proc. of the AAAI Conference on Artificial Intelligence*, 2006, pp. 913–919.

[8] J. Kucera and R. Barták, "LOUGA: learning planning operators using genetic algorithms," in *In Proc. of the Pacific Rim Knowledge Acquisition Workshop*, 2018, pp. 124–138.

[9] K. Mourão, L. S. Zettlemoyer, R. P. A. Petrick, and M. Steedman, "Learning STRIPS operators from noisy and incomplete observations," in *In Proc. of the Conference on Uncertainty in Artificial Intelligence*, 2012, pp. 614–623.

[10] S. Cresswell, T. L. McCluskey, and M. M. West, "Acquiring planning domain models using *LOCM*," *Knowledge Engineering Review*, vol. 28, no. 2, pp. 195–213, 2013.

[11] C. Rodrigues, P. Gérard, and C. Rouveirol, "Incremental learning of relational action models in noisy environments," in *In Proc. of the International Conference on Inductive Logic Programming*, 2010, pp. 206–213.

[12] J. Á. Segura-Muros, R. Pérez, and J. Fernández-Olivares, "Learning numerical action models from noisy and partially observable states by means of inductive rule learning techniques," in *In Proc. of the workshop on Scheduling and Knowledge Engineering for Planning and Scheduling*, 2018, pp. 46–53.

[13] M. Grand, H. Fiorino, and D. Pellier, "Amlsi: A novel and accurate action model learning algorithm," in *In Proc. of the International Workshop on Knowledge Engineering for Planning and Scheduling*, 2020.

[14] E. M. Gold, "Language identification in the limit," *Information and Control*, vol. 10, no. 5, pp. 447–474, 1967.

[15] J. L. Bresina, A. K. Jónsson, P. H. Morris, and K. Rajan, "Activity planning for the mars exploration rovers," in *Proc. of the International Conference on International Conference on Automated Planning and Scheduling*, 2005, p. 40–49.

[16] Y. Carreno, E. Pairet, Y. Petillot, and R. P. A. Petrick, "Task allocation strategy for heterogeneous robot teams in offshore missions," in *In Proc. of the International Conference on Autonomous Agents and MultiAgent Systems*, 2020, p. 222–230.

[17] D. Höller, G. Behnke, P. Bercher, and S. Biundo, "Assessing the expressivity of planning formalisms through the comparison to formal languages," in *In Proc. of the International Conference on Planning and Scheduling*, 2016, p. 158–165.

[18] J. Oncina and P. García, "Inferring regular languages in polynomial update time," in *Pattern Recognition and Image Analysis: Selected Papers from the IVth Spanish Symposium*. World Scientific, 1992, vol. 1, pp. 49–61.

[19] M. Grand, H. Fiorino, and D. Pellier, "Retro-engineering state machines into pddl domains," in *In Proc. of the International Conference on Tools with Artificial Intelligence*, 2020, pp. 1186–1193.

[20] H. H. Zhuo, Q. Yang, D. H. Hu, and L. Li, "Learning complex action models with quantifiers and logical implications," *Artificial Intelligence*, vol. 174, no. 18, pp. 1540–1569, 2010.

[21] H. H. Zhuo, T. A. Nguyen, and S. Kambhampati, "Refining incomplete planning domain models through plan traces," in *In Proc. of the International Joint Conference on Artificial Intelligence*, 2013, pp. 2451–2458.

[22] M. Helmert, "The Fast Downward planning system," *Artificial Intelligence*, vol. 26, pp. 191–246, 2006.

[23] R. Howey and D. Long, "Val's progress: The automatic validation tool for pddl2. 1 used in the international planning competition," in *In Proc. of the International Workshop on the International Planning Competition*, 2003, pp. 28–37.

[24] T. L. McCluskey, N. E. Richardson, and R. M. Simpson, "An interactive method for inducing operator descriptions." in *In Proc. of the Artificial Intelligence Planning and Scheduling Conference*, 2002, pp. 121–130.

[25] X. Wang, "Learning by observation and practice: An incremental approach for planning operator acquisition," in *In Proc. of the International Conference on Machine Learning*, 1995, pp. 549–557.

[26] S. Cresswell and P. Gregory, "Generalised domain model acquisition from action traces," in *In Proc. of the International Conference on Planning and Scheduling*, 2011, pp. 42–49.

[27] P. Gregory and S. Cresswell, "Domain model acquisition in the presence of static relations in the LOP system," in *In Proc. of the International Conference on Planning and Scheduling*, 2015, pp. 97–105.

[28] P. Gregory and A. Lindsay, "Domain model acquisition in domains with action costs," in *In Proc. of the International Conference on Planning and Scheduling*, 2016, pp. 149–157.