An Accurate HDDL Domain Learning Algorithm from Partial and Noisy Observations

Maxence Grand, Damien Pellier, Humbert Fiorino

Univ. Grenoble Alpes, LIG, Grenoble, France
{Maxence.Grand, Damien.Pellier, Humbert.Fiorino}@univ-grenoble-alpes.fr

Abstract

The Hierarchical Task Network (HTN) formalism is very expressive and used to express a wide variety of planning problems. In contrast to the classical STRIPS formalism in which only the action model needs to be specified, the HTN formalism requires to specify, in addition, the tasks of the problem and their decomposition into subtasks, called HTN methods. For this reason, hand-encoding HTN problems is considered more difficult and more error-prone by experts than classical planning problem. To tackle this problem, we propose a new approach (HierAMLSI) based on grammar induction to acquire HTN planning domain knowledge, by learning action models and HTN methods with their preconditions. Unlike other approaches, HierAMLSI is able to learn both actions and methods with noisy and partial inputs observation with a high level or accuracy.

1 Introduction

The Hierarchical Task Network (HTN) formalism (Erol, Hendler, and Nau 1994) is very expressive and used to express a wide variety of planning problems. This formalism allows planners to exploit domain knowledge to solve problems more efficiently (Nau et al. 2005) when planning problems can be naturally decomposed hierarchically in terms of tasks and task decompositions. The standard language used to model HTN problem is HDDL (Hierachical Domain Description Language) (Höller et al. 2020). In contrast to the classical PDDL language used to model STRIPS problems in which only the action model needs to be specified, HDDL requires to specify the task model of the problem. A task model can be primitive and compound. A primitive task model is described by PDDL operators. A compound tasks model is described using HTN methods. An HTN method describes the set of primitive and/or compound task required to decompose a specific compound task. For this reason, hand-encoding HTN problems is considered more difficult and more error-prone by experts than classical planning problem. This makes it all the more necessary to develop techniques to learn HTN domains.

Many machine learning approaches have been proposed to facilitate the acquisition of PDDL domain acquisition and to learn the underlying action model, e.g, ARMS (Yang, Wu, and Jiang 2007), FAMA (Aineto, Celorrio, and Onaindia 2019), LOCM (Cresswell, McCluskey, and West 2013),

LSONIO (Mourão et al. 2012), AMLSI (Grand, Fiorino, and Pellier 2020a,b). In these approaches, training data are either (possibly noisy and partial) intermediate states and plans previously generated by a planner, or randomly generated action sequences (i.e. random walks). On the other hand, few approaches have been proposed to learn HTN domains. However, it is possible to mention CAMEL (Ilghami et al. 2002), HTN-Maker (Hogg, Munoz-Avila, and Kuter 2008; Hogg, Kuter, and Munoz-Avila 2009), LHTNDT (Nargesian and Ghassem-Sani 2008) or HTN-Learner (Zhuo et al. 2009). The major drawbacks of these approaches are: (1) they consider to have complete and noiseless observations as input; (2) they only learn HTN methods except HTN-Learner, i.e., they consider that the action model is known a priori and (3) the learned domains are not *accurate* enough to be used "as is" in a planner. A step of expert proofreading is still necessary to correct them. Even small syntactical errors can make sometime the learned domains useless for planning

To deal with these drawbacks, we propose in this paper, a new learning algorithm for HDDL domains, called HierAMLSI. HierAMLSI is based on AMLSI (Grand, Fiorino, and Pellier 2020a,b), a PDDL domain learner based on grammar induction. HierAMLSI takes as input a set of partial and noisy observations and learns a full HDDL planning domain with action model and HTN methods. We show experimentally that HierAMLSI is highly accurate even with highly partial and noisy learning datasets minimising HDDL domain proofreading by experts. In many HDDL ICP benchmarks HierAMLSI does not require any correction of the learned domains at all.

The rest of the paper is organized as follows. In section 2 we present the problem statement. In section 3 we give some backgrounds on the AMLSI approach. In section 4, we detail the HierAMLSI steps. Then, section 5 evaluates the performance of HierAMLSI on IPC benchmarks. Finally, Section 6 concludes with the related works.

2 Formal Framework

Section 2.1 introduces a formalization of STRIPS planning domain learning consisting in learning a transition function of a grounded planning domain and in expressing it as PDDL operators and Section 2.2 extends this formalization to HTN domains.

2.1 STRIPS Planning

In this section we use definitions and notations proposed by (Höller et al. 2016) and adapt them to the learning problem.

A STRIPS planning problem is a tuple $P = (L, A, S, s_0, s, \delta, \lambda)$, where L is a set of logical propositions describing the world states, S is a set of state labels, $s_0 \in S$ is the label of the initial state, and $g \subseteq S$ is the set of goal labels. λ is an observation function $\lambda : S \to 2^L$ that assigns to each state label the set of logical propositions true in that state. A is a set of action labels. Action preconditions, positive and negative effects are given by the functions $prec$, add and del that are included in $\delta = (prec, add, del)$. $prec$ 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 (Höller et al. 2016) in order to define the STRIPS learning problem as the lifting of a state transition system into a propositional language.

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 action a is applicable in state s_i, the state transition function $\gamma : A \times S \to S$ returns the resulting state $s_{i+1} = \gamma(s_i, a)$ such that $\lambda(s_{i+1}) = [\lambda(s_i) \setminus del(a)] \cup add(a)$.

A sequence $(a_0 a_1 \ldots a_n)$ 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 $(a_0 a_1 \ldots a_n)$ in state s_0, $\gamma(s_0, (a_0 a_1 \ldots a_n)) = \gamma(\gamma(s_0, a_0), (a_1 \ldots a_n)) = s_{n+1}$. It is important to note that this recursive definition of γ entails the generation of a sequence of states $(s_0 s_1 \ldots s_{n+1})$. A goal state is a state s such that $g \in G$ and $\lambda(g) \subseteq \lambda(s)$. s satisfies g, i.e. $s \models g$ if and only if s is a goal state. An action sequence is a solution plan to a planning problem P if and only if it is applicable to s_0 and entails a goal state.

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 ($0 \leq i \leq n$):

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

We know that the set of languages generated by STRIPS planning problems are regular languages (Höller et al. 2016). In other words, a STRIPS planning problem $P = (L, A, S, s_0, G, \delta, \lambda)$ generates 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 is the transition function.

For any arc $a \in A$, we call *pre-set* of a the set $preset(a) = \{s \in S \mid \gamma(s, a) = s'\}$ and *post-set* of a the set $postset(a) = \{s' \in S \mid \gamma(s, a) = s'\}$ (see Figure 1).

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

For instance, suppose $\Omega = \{a, ab, ba, bab, abb, \ldots\}$ such that $s_0 \xrightarrow{a} s_2$, $s_0 \xrightarrow{a} s_2 \xrightarrow{b} s_2$, $s_0 \xrightarrow{b} s_1 \xrightarrow{a} s_2$, $s_0 \xrightarrow{b} s_1 \xrightarrow{a} s_2 \xrightarrow{b} s_2$, $s_0 \xrightarrow{a} s_2 \xrightarrow{b} s_2 \xrightarrow{b} s_2 \ldots$ (Grand, Fiorino, and Pellier 2020b) show that it is possible to learn Σ (see Figure 1)
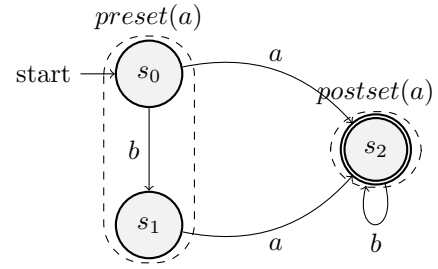

Figure 1: An example of DFA with pre-states and post-states

and infer P with actions $\{a, b\}$, the initial state s_0 and some states marked as goal $G = \{s_2\}$.

2.2 HTN Planning

By extension based on the notation of (Höller 2021), an HTN planning problem is a tuple $P = (L, C, A, S, M, s_0, w_I, g, \delta, \tau, \lambda, \sigma, \zeta)$. As for STRIPS problems, L is a set of logical propositions describing the world states, S is a set of state labels, $s_0 \in S$ is the label of the initial state, $g \subseteq S$ is the set of goal label, λ is the observation function and preconditions, positive and negative effects are given by the functions $prec$, add and del included in δ.

A is the set action (or primitive task) labels and C is a set of compound (or non primitive) task labels, with $C \cap A = \emptyset$. Tasks are maintained in *task networks*. A task network is a sequence of tasks (for simplicity, we consider only Totally Ordered domain). Let $T = C \cup A$. A task network is an element out of T^* (* is the Kleene operator). Compound tasks are decomposed using methods. The set M contains all method labels. Methods are defined by the function $\sigma : M \to C \times T^*$. Then, a coumpound task c is decomposable in a state s if and only if there exists a revelant method $m \in M$ such that: $\sigma(m) = (c, \phi)$ and $prec(m) \in s$. The function $\zeta : T^* \times S \to T^*$ gives the decomposition function. For a totally orderer task network $\omega = \omega_1 t \omega_2$, ζ is defined as follows:

$$\zeta(\omega_1 t \omega_2, s) = \begin{cases} \omega_1 t \omega_2 & \text{if } t \text{ is a primitive task} \\ \omega_1 \phi \omega_2 & \text{if } t \text{ is a coumpound task} \\ & \text{and } t \text{ is decomposable in } \gamma(\omega_1, s) \\ \emptyset & \text{Otherwise} \end{cases}$$

As $\omega_1 t \omega_2$ is a totally ordered task network, ω_1 contains only primitive tasks. We denote $\omega \to^* \omega'$ that ω can be decomposed into ω' by 0 or more method applications. Finally, ω_I is the initial task network.

A solution to an HTN planning problem is a task network ω with:

1. $\omega_I \to^* \omega$, i.e. it can be reached by decomposing ω_I.

2. $\omega \in A^*$, i.e. all tasks are primitive.

3. $\gamma(s_0, \omega) \models g$, i.e. ω is applicable in s_o and results in a goal state.

Finally, we can define an HTN planning problem $P = (L, C, A, S, M, s_0, w_I, g, \delta, \tau, \lambda, \sigma, \zeta)$ as a formal language:

$$\mathcal{L}(P) = \{\omega = (t_0 t_1 \ldots t_n) | t_i \in A, \gamma(s_0, \omega) \models g, \omega_I \to^* \omega\}$$

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

Unlike STRIPS planning problems, the language $\mathcal{L}(P)$ is not necessary regular (Höller et al. 2014) and cannot be represented as a DFA. As mentioned by (Höller et al. 2014; Höller 2021), $\mathcal{L}(P)$ is the intersection of two languages:

1. $\mathcal{L}_{\mathcal{H}}(P) = \{\omega \in A^* | w_I \rightarrow^* \omega\}$ which is defined by the decomposition hierarchy, i.e. by the compound tasks and methods.

2. $\mathcal{L}_{\mathcal{C}}(P) = \{\omega \in A^* | \gamma(s_0, \omega) \in g\}$ which is defined by the state transition system defined by the preconditions and effects of the primitive tasks. This language is regular.

The key idea of our approach is to learn the DFA $\Sigma = (S, A, \gamma)$ corresponding to the regular language $\mathcal{L}_{\mathcal{C}}(P)$, and modify the DFA to approximate the language $\mathcal{L}(P)$ with the DFA $\Sigma = (S, T, \gamma)$ and then infer P.

3 The AMLSI Approach

In this section we will summarized the AMLSI approach on which HierAMLSI is based. For more detail see (Grand, Fiorino, and Pellier 2020b,a).

AMLSI generates the set of observations Ω by using random walks 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 observation is a state where the truth value of a proposition is erroneous). No knowledge of the goal states G is required. Once Σ is learned, AMLSI has to deduce δ from the transition function γ. Concretely, δ can be represented as a STRIPS 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) induction of the PDDL operators from the learned DFA; (4) finally, refinement of these operators to deal with noisy and partial state observations:

Step 1: AMLSI generates a 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.

Step 2: To learn the DFA $\Sigma = (S, A, \gamma)$ AMLSI uses a variant (Grand, Fiorino, and Pellier 2020b) of a classical regular grammar learning algorithm called RPNI (Oncina and García 1992). The learning is based on both I^+ and I^-.

Step 3: AMLSI begins by inducing the preconditions and effects of the actions. For the preconditions $prec(a)$ of action a, AMLSI computes the logical propositions that are in all the states preceding a in Σ:

$$prec(a) = \cap_{s \in preset(a)} \lambda(s)$$

For the positive effects $add(a)$ of action a, AMLSI computes the logical propositions that are never in states before the execution of a, and always present after a execution:

$$add(a) = \cap_{s \in preset(a)} \lambda(s) \setminus prec(a)$$

Dually,

$$del(a) = prec(a) \setminus \cap_{s \in postset(a)} \lambda(s)$$

Once preconditions and effects are induced, actions are lifted to PDDL operators based on OI-subsumption (subsumption under Object Identity) (Esposito et al. 2000): first of all, constant symbols in preconditions and effects are substituted by variable symbols. Then, the less general preconditions and effects, i.e. preconditions and effects encoding as many propositions as possible, are computed as intersection sets. This generalization method allows to ensure that all the necessary preconditions, i.e. the preconditions allowing to differentiate the states where actions are applicable from states where they are not, to be rightfully coded in the corresponding operators.

Step 4: To deal with noisy and partial state observations, 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 the effects ensuring that each transition in the DFA are feasible. Then, AMLSI refines the preconditions of the operators. As in (Yang, Wu, and Jiang 2007), it makes the following assumptions: 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.

4 The HierAMLSI approach

HierAMLSI generates the set of observations Ω by using random walks to learn $\Sigma = (S, T, \gamma)$ and deduce $P = (L, C, A, S, M, s_0, w_I, g, \delta, \tau, \lambda, \sigma, \zeta)$. HierAMLSI makes the same assumptions than AMLSI: L, A, S, s_0 are known but also C, the decomposition function ζ and the observation function λ possibly partial and noisy (a partial observation is a state where some propositions are missing and a noisy observation is a state where the truth value of a proposition is erroneous). No knowledge of the goal states g is required. Once Σ is learned, HierAMLSI has to deduce δ from the transition function γ and σ from the decomposition function ζ.

Like AMLSI, HierAMLSI uses random walks to generate Ω and makes the same assumptions.

HierAMLSI has been devised to solve HTN learning problems. In practice, it computes P as HDDL domain and problem files (Höller et al. 2020; Höller et al. 2019).

Regarding the training dataset, HierAMLSI uses random walks to generate Ω. HierAMLSI makes the same assumptions than AMLSI. Once the DFA is learned, HierAMLSI generates the set of methods σ and infers the action precondition, positive and negative effect functions in δ from the state transition function γ. Finally, the methods and operators of the HDDL domain file are induced from σ and δ.

The HierAMLSI approach consists of 4 steps:

1. *Generation of the observations.* HierAMLSI produces a set of observations Ω by using a random walk. In section 4.1, we will present how HierAMLSI is able to efficiently exploit these observations by taking into account not only the fact that some task are decomposable in certain states and their decomposition but also that others are not.

2. *DFA Learning.* HierAMLSI learns a DFA approximating the language $\mathcal{L}(P)$ (see Section 4.2).

3. HTN *Methods generation.* HierAMLSI generates from the DFA learned previously a set of HTN Methods allowing to decompose all tasks observed in Ω (see Section 4.3).

4. *Action model and* HTN *Methods precondition learning.* Once HTN Methods have been learned, HierAMLSI have to learn the action model, i.e. primitive tasks preconditions and effects and the HTN Methods preconditions. To do this, HierAMLSI treat HTN Methods as primitive tasks and learn an action model containing both methods and primitive tasks using the learning and refinement techniques proposed by the AMLSI approach (see Section 3).

The rest of this Section will be illustrated using the IPC[1] Gripper domain. In this domain, a robot moves balls in different rooms using its two grippers r and l. This domain contains three compound tasks: $goto(r_a)$ the robot goes into the room r_a, $move1ball(b_1\ r_a)$ the robots move the ball b_1 in the room r_a and $move2balls(b_1\ b_2\ r_a)$ the robot moves balls b_1 and b_2 in the room r_a.

4.1 Observation Generation

The data generation process is similar to the generation method of the AMLSI algorithm (Grand, Fiorino, and Pellier 2020b). To generate the observations in Ω, HierAMLSI uses random walks by querying a blackbox. HierAMLSI chooses randomly a (primitive or compound) task t. If the task t is decomposable, the blackbox returns the final decomposition ϕ containing only primitive task and this decomposition is added to the current primitive task sequence. This procedure is repeated until the selected task is not applicable to the current state. The applicable prefix of the primitive task sequence is then added to I_+, the set of positive samples, and the complete sequence, whose last task is not applicable, is added to I_-, the set of negative samples. Random walks are repeated until I_+ and I_- achieve an arbitrary size.

4.2 DFA Learning

As mentioned in Section 2 the language $\mathcal{L}(P)$ is not necessary regular, then the purpose of this step is to learn a DFA

[1]https://www.icaps-conference.org/competitions/

Algorithm 1: Heuristic HTN Methods learning

1   $Initialization(M)$
2   **for** $i = 1 : |C|$ **do**
3   **for** $c \in C$ **do**
4       $M'[c] \leftarrow greedy(c, M, i)$
5   **for** $c \in C$ **do**
6   **if** $|M'[c]| < |M[c]|$ **then**
7         $M[c] \leftarrow M'[c]$
8   **return** M

---

approximating this language. More precisely, the DFA learning step is divided in 2 steps: (1) HierAMLSI learns a DFA corresponding to the language $\mathcal{L}_{\mathcal{C}}(P)$ which is defined by the state transition system defined by the preconditions and effects of the primitive tasks and (2) HierAMLSI adds transitions to represent compound tasks in the DFA to allow to approximate the language $\mathcal{L}(P)$.

Step 1: Primitive task DFA Learning   HierAMLSI starts by using the DFA Learning algorithm proposed by (Grand, Fiorino, and Pellier 2020b) to learn the DFA containing only primitive tasks.

Step 2: Task DFA Induction   Once the Primitive Task DFA has been learned, HierAMLSI induces the Task DFA by adding compound task transition in the DFA, i.e. by adding transitions whose labels are compound task labels. For instance, suppose we have the compound task $move2balls(b_1\ b_2\ r_b)$ has been decomposed by primitive tasks $(move(r_a\ r_b),\ pick(b_1\ r\ r_b),\ pick(b_2\ l\ r_b),\ move(r_b\ r_a),\ drop(b_1\ r\ r_a),\ drop(b_2\ l\ r_a))$ in node 0 and reached the node 6. Then we add the following transitions in the DFA $\gamma(move2balls(b_1\ b_2\ r_b), 0) \rightarrow 6$. Figure 2 gives an example of Task DFA.

4.3 HTN Methods Learning

Once the Task DFA is induced HierAMLSI can directly extract HTN Methods from the Task DFA. However, it is possible that a large number of HTN Methods has been generated. For instance, let's take the Task DFA in Figure 2. For the compound task $move1ball(?b\ ?r)$ HierAMLSI can generate several methods:

$\phi_1 = (move(?r\ ?r_1), pick(?b\ ?g\ ?r_1), move(?r_1\ ?r), drop(?b\ ?g\ ?r))$
$\phi_2 = (pick(?b\ ?g\ ?r_1), move(?r_1\ ?r), drop(?b\ ?g\ ?r))$
$\phi_3 = (goto(?r_1), pick(?b\ ?g\ ?r_1), move(?r_1\ ?r), drop(?b\ ?g\ ?r))$
$\phi_4 = (goto(?r_1), pick(?b\ ?g\ ?r_1), goto(?r), drop(?b\ ?g\ ?r))$
$\cdots$

Some of these decomposition are redundant. To facilitate proof reading we want a more compact description of the HTN Methods. More precisely, we want minimizing the set of methods allowing to decompose observed compound tasks. Then, the HTN Methods learning problem can be reduce to a variant of the set cover problem (Karp 1972) which is NP-Complete.

The Greedy algorithm (Jungnickel 1999) is a classical way to approximate the solution in a polynomial time. At each stage of the Greedy algorithm, HierAMLSI chooses
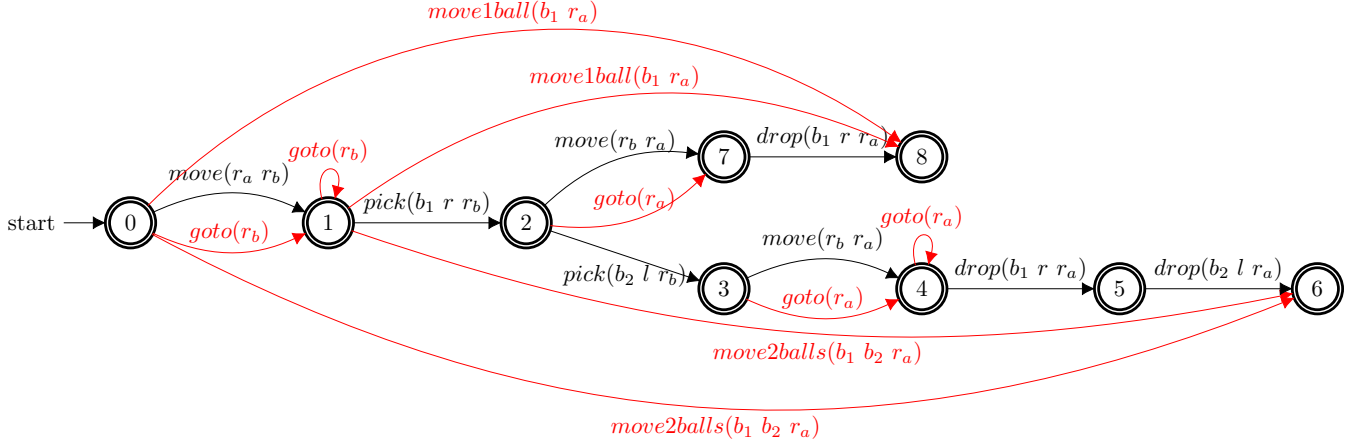
Figure 2: DFA learning steps The Primitive Task DFA is the DFA containing only Primitive Task, i.e. black transition, and the Task DFA contains Compound Tasks, i.e. red transitions, in addition
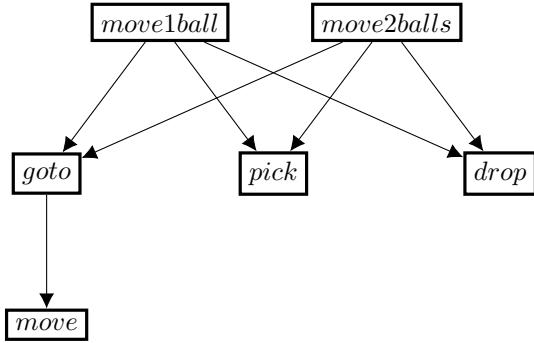

Figure 3: Tasks dependencies for the IPC Gripper domain

the method that allows to decompose the largest number of tasks.

The main drawback of this approach is that it does not take into account the fact that there are dependencies between tasks. For instance, the optimal way to decompose the compound task $move1ball(?b\ ?r)$ is $\phi_4 = (goto(?r_1)$, $pick(?b\ ?g\ ?r_1)$, $goto(?r)$, $drop(?b\ ?g\ ?r))$ (see Figure 3). So the compound task $move1ball(?b\ ?r)$ depends of the compound task $goto(?r)$. So, as long as all methods for the compound task $goto(?r)$ have been generated, the Greedy algorithm will always prioritize ϕ_1 and ϕ_2 to ϕ_4. Indeed, the decomposition ϕ_4 can be added to the current solution of the Greedy algorithm if and only if all methods for the compound task $goto(?r)$ have been added.

Heuristic Approach We propose a sound, complete and polynomial Heuristic approach (see Algorithm 1) taking into account dependencies between tasks. Figure 4 gives an example for the Gripper domain.

HierAMLSI starts by initializing the set of HTN methods using the decomposition ϕ observed during the observation generation step (see Section 4.1). For each Compound Task

we have therefore a set of HTN Methods containing only Primitive Tasks and no Compound Task dependencies. For instance, for the Compound Task $move1ball(?b\ ?r)$ we have the two following decomposition:

$$\phi_1 = (pick(?b\ ?g\ ?r_1), move(?r_1\ ?r), drop(?b\ ?g\ ?r))$$
$$\phi_2 = (move(?r_?\ ?r_1)), pick(?b\ ?g\ ?r_1), move(?r_1\ ?r), drop(?b\ ?g\ ?r))$$

Then, at each iteration AMLSI use the greedy algorithm to compute a new set of HTN Methods with an additional Compound Task dependency. Finally, if the new HTN Methods set is smaller than the one learned in the previous iteration, then it is retained. For instance, for the Gripper domain, suppose we have the two following decomposition for the Compound Task $goto(?r)$:

$$\phi_1 = ()$$
$$\phi_2 = (move(?r_1\ ?r)))$$

Then, the Greedy algorithm return only one decomposition for the Compound Task $move1ball(?b?r)$: $(goto(?r_1))$, $pick(?b\ ?g\ ?r_1)$, $goto(?r)$, $drop(?b\ ?g\ ?r))$.

Lemma 1. *The Heuristic approach is sound and complete. The heuristic approach generates a set of* HTN *Methods M able to decompose all observed compound tasks in the dataset Ω.*

Proof. During the observation generation step (see Section 4.1), for each generated Compound Task t, we have its final decomposition ϕ. So at the initialization step of the Heuristic approach, there at least one method able to decompose each observed Task. The initialization is therefore sound and complete. Moreover the following steps of the Heuristic approach generates methods decomposing as many tasks as the previous steps, then the Heuristic approach is sound and complete. □

Lemma 2. *The Heuristic approach is polynomial.*

$$goto(?r):\qquad\qquad \phi_1 = (), \phi_2 = move(?r_1\ ?r)$$
$$move2balls(?b_1\ ?b_2\ ?r):\quad \phi_1 = (pick(?b_1\ ?g_1\ ?r_1), pick(?b_1\ ?g_1\ ?r_1), move(?r_1\ ?r), drop(?b_1\ ?g_1\ ?r), drop(?b_1\ ?g_1\ ?r))$$
$$\phi_2 = (move(?r\ ?r_1), pick(?b_1\ ?g_1\ ?r_1), pick(?b_1\ ?g_1\ ?r_1), move(?r_1\ ?r), drop(?b_1\ ?g_1\ ?r), drop(?b_1\ ?g_1\ ?r))$$
$$move1ball(?b\ ?r):\qquad\quad \phi_1 = (pick(?b\ ?g\ ?r_1), move(?r_1\ ?r), drop(?b\ ?g\ ?r))$$
$$\phi_2 = (move(?r_?\ ?r_1)), pick(?b\ ?g\ ?r_1), move(?r_1\ ?r), drop(?b\ ?g\ ?r))$$

(a) **Step 0:** Initialization with no compound task dependency

$$goto(?r):\qquad\qquad \phi_1 = (), \phi_2 = move(?r_1\ ?r)$$
$$move2balls(?b_1\ ?b_2\ ?r):\quad \phi_1 = (goto(?r_1), pick(?b_1\ ?g_1\ ?r_1), pick(?b_1\ ?g_1\ ?r_1), goto(?r), drop(?b_1\ ?g_1\ ?r), drop(?b_1\ ?g_1\ ?r))$$
$$move1ball(?b\ ?r):\qquad\quad \phi_1 = (goto(?r_1), pick(?b\ ?g\ ?r_1), goto(?r), drop(?b\ ?g\ ?r))\}$$

(b) **Step 1:** Induction with 1 compound task dependence

$$goto(?r):\qquad\qquad \phi_1 = (), \phi_2 = move(?r_1\ ?r)$$
$$move2balls(?b_1\ ?b_2\ ?r):\quad \phi_1 = (goto(?r_1), pick(?b_1\ ?g_1\ ?r_1), pick(?b_1\ ?g_1\ ?r_1), goto(?r), drop(?b_1\ ?g_1\ ?r), drop(?b_1\ ?g_1\ ?r))$$
$$move1ball(?b\ ?r):\qquad\quad \phi_1 = (goto(?r_1), pick(?b\ ?g\ ?r_1), goto(?r), drop(?b\ ?g\ ?r))\}$$

(c) **Step n:** Induction with n compound task dependence

Figure 4: HTN Methods learning example

Domain	# Primitive Task	# Compound Task	# Methods	# Predicates
Blocksworld	4	4	8	5
Gripper	3	3	4	4
Zenotravel	4	2	5	7
Transport	3	4	6	5
Childsnack	6	1	2	12

Table 1: Benchmark domain characteristics. From left to right, the number of Primitive Tasks, the number of Compound Tasks, the number of Methods and the number of Predicates for each IPC domain.

Proof. First of all, we have $\mathcal{O}(|I_+|)^2$ nodes in the DFA. Then, in the worst case, we have a possible HTN Method for each node pair, then we have $\mathcal{O}(|I_+|^2)$ possible HTN Methods in the DFA. Then, the complexity of the Greedy algorithm is $\mathcal{O}(|I_+|^3)$ in term of tested decomposition. Finally, according to the algorithm 1, the Greedy algorithm is repeated $|C|^2$ times. Finally, the complexity of the Heuristic approach is $\mathcal{O}(|C|^2.|I_+|^3)$. □

5 Exprimentation

The purpose of this evaluation is to evaluate the performance of HierAMLSI though two variants: (1) we evaluate the performance of HierAMLSI when only HTN Methods are learned, i.e. the action model is known and (2) we evaluate the performance of HierAMLSI when both HTN Methods are learned and the action model is unknown. We use 4 experimental scenarios[3]:

1. Complete intermediate observations (100%) and no noise (0%).
2. Complete intermediate observations (100%) and high level of noise (20%).

---

[2]$|I_+|$ denote the number of primitive tasks in the positive sample

[3]Note that these are the experimental scenarios used to test AMLSI on which HierAMLSI is built

3. Partial intermediate observations (25%) and no noise (0%).
4. Partial intermediate observations (25%) and high level of noise (20%).

5.1 Experimental Setup

Our experiments are based on 5 HDDL (Höller et al. 2020; Höller et al. 2019) domains (see Table 1) from the IPC 2020 competition: Blocksworld, Childsnack, Transport, Zenotravel and Gripper.

HierAMLSI learns HTN domains from one instance. To avoid performances being biased by the initial state, HierAMLSI is evaluated with different instances. Also, for each instance, to avoid performances being biased by the generated observations, experiments are repeated five times. 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. PDDL4J library (Pellier and Fiorino 2018) was used to generate the benchmark data.

5.2 Evaluation Metrics

HierAMLSI is evaluated using the *accuracy* (Zhuo, Nguyen, and Kambhampati 2013) that measures the learned domain performance to solve new 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. In the rest of this section the accuracy is computed over 20 problems. The problems are solved with the TFD planner (Pellier and Fiorino 2020) provided by the PDDL4J library. Plan validation is done with VAL, the IPC competition validation tool (Howey and Long 2003).

5.3 Discussion

Table 5 shows the accuracy of HierAMLSI obtained on the 5 domains of our benchmarks when varying the training data
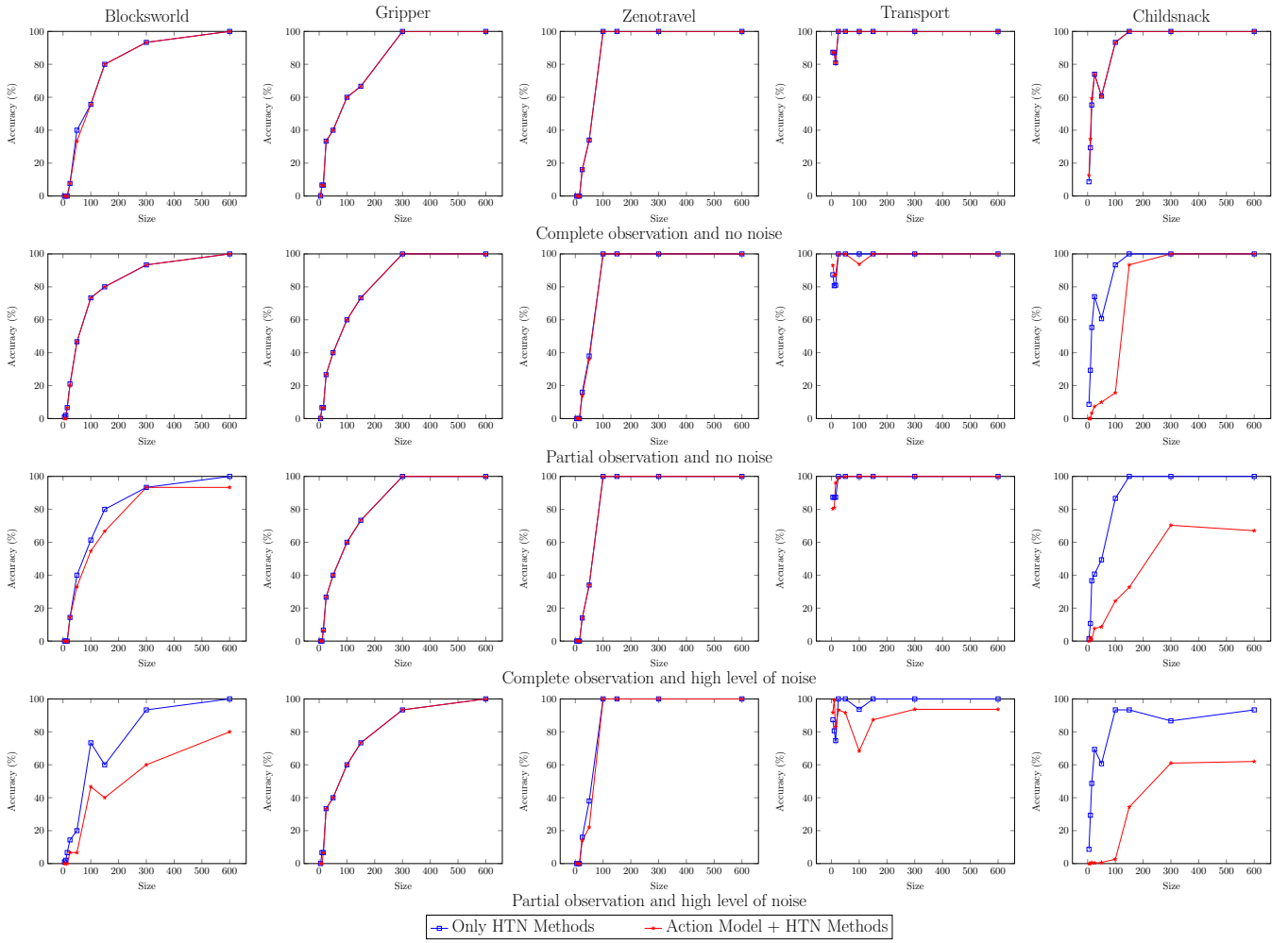
Figure 5: Performance of HierAMLSI when the training data set size increases in number of tasks in terms of Accuracy.

set size. The size of the training set is indicated in number of tasks.

First of all, we observe that when HierAMLSI learns only the set of HTN Methods, learned domains are generally optimal (Accuracy = 100%) with 600 tasks whatever the experimental scenario. Also, 100 tasks are generally sufficient to learn accurate domains (Accuracy > 50%). Then, when HierAMLSI learns both action model and HTN Methods performances are similar when observations are noiseless. However, when observations are noisy, performances are downgraded for some domains: Blocksworld and Childsnack when observations are complete and Blocksworld, Transport and Childsnack when observations are partial. This is due to the fact that there are learning error in the action model learned. However, learned domains remain accurate when there are at least 300 tasks in the training dataset.

To conclude, we have shown experimentally that Hier-AMLSI learns accurate domains. More precisely, when the action model is known, HierAMLSI generally learns optimal domains. Also the performances are downgraded when HierAMLSI has to learn the action model in addition to the set of HTN methods, but the learned domains remain accurate. Performance degradation are due to the learning errors in the action model.

6 Related Works

Many approaches have been proposed to learn HDDL domains. These approaches can be classified according to the input data of the learning process. The input data can be plan "traces" obtained by resolving a set of planning problems, annotated plans (see Figure 6a), decomposition tree (see Figure 6b) or random walks. The input data can contain in addition to the tasks also states which can be fully observable (FO) or partially observable (PO), or noisy. Also, these approaches can be classified according to the output. The output can be the action model, the set of HTN Methods and HTN Methods preconditions. Table 2 summarises these classifications.

A first group of approaches only learns the set of HTN Methods. First of all, (Xiaoa et al. 2019) takes as input a set of plan traces and HTN Methods and proposes an algorithm to update incomplete HTN Methods by task insertions.

Algorithm	Input			Output		
	Input	Environment	Noise Level	Action Model	HTN Methods	HTN Methods Preconditions
(Xiaoa et al. 2019)	Plans	FO	0%		•	
HTN-Maker	Plans	FO	0%		•	
(Hogg, Kuter, and Munoz-Avila 2010)	Plans	FO	0%		•	
HTN-MakerND	Plans	FO	0%		•	
(Li et al. 2014)	Plans	FO	0%		•	
(Garland and Lesh 2003)	Annotated Plans	PO	0%		•	
LHTNDT	Annotated Plans	FO	0%		•	
CAMEL	Plans	FO	0%		•	•
HDL	Plans	FO	0%		•	•
HTN-Learner	Decomposition Trees	PO	0%	•	•	•
HierAMLSI	Random Walks	PO	20%	•	•	•

Table 2: State-of-the-art action model learning algorithms. From left to right: the kind of input data, the kind of environment: **F**ully **O**bservable or **P**artially **O**bservable, the maximum level of noise in observations, the kind of output

(a) An annotated plan
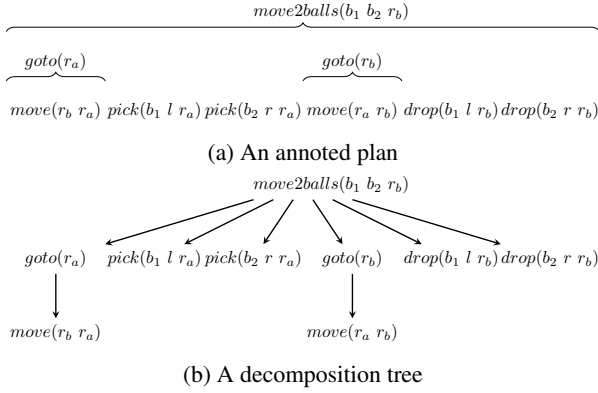

(b) A decomposition tree

Figure 6: Input examples

Then HTN-Maker (Hogg, Munoz-Avila, and Kuter 2008) and HTN-MakerND (Hogg, Kuter, and Munoz-Avila 2009) takes as input plan trace generated from STRIPS planner and annotated task provided by a domain expert. Then, (Hogg, Kuter, and Munoz-Avila 2010) proposed an algorithm based on reinforcement learning. Then, (Li et al. 2014) proposed an algorithm taking as input only plan traces. This algorithm builds, from plan traces, a context free grammar (CFG) allowing to regenerate all plans. Then, methods are generated using CFG: one method for each production rule in the CFG. Then (Garland and Lesh 2003) and (Nargesian and Ghassem-Sani 2008) proposed to learn HTN Methods from annotated plan. Annotated plan are plan segmented with the different tasks solved. Figure 6a gives an example of annotated plan. However, obtaining these annotated examples is difficult and needs a lot of human effort.

A second group of approach learns HTN Methods preconditions. First of all, the CAMEL algorithm (Ilghami et al. 2002) learns HTN Methods and their the preconditions of HTN Methods from observations of plan traces, using the version space algorithm. An annotated task is an triplet (n, Pre, Eff) where n is a task, Pre is a set of proposition known as the preconditions and Eff is a set of atoms known as the effects. These approach use annotated task to build incrementally HTN Methods with preconditions. Then, the HDL algorithm (Ilghami, Nau, and Muñoz-Avila

2006) takes as input plan traces. For each decomposition in plan traces, HDL checks if there exist a method responsible of this decomposition. If not, HDL creates a new method and initializes a new version space to capture its preconditions. Preconditions are learned in the same way as in the CAMEL algorithm.

Only HTN-Learner proposes to learn Action Model and HTN Methods from decomposition trees. A decomposition tree is a tree corresponding to the decomposition of a method. Figure 6b gives an example of decomposition tree.

7 Conclusion

In this paper we have presented HierAMLSI, a novel algorithm to learn HDDL domains. HierAMLSI is built on the AMLSI approach. HierAMLSI is composed of four steps. The first step consists, as AMLSI, in building two training sets of feasible and infeasible action sequences. In the second step, HierAMLSI induces a DFA. The third step is the generation of the HTN Methods, and the last step learns HTN Methods preconditions and the action model. Our experimental results show that HierAMLSI is able to learn accurately both action models and HTN Methods from partial and noisy datasets.

Future works will focus on extending HierAMLSI in order to learn more expressive action model.

Acknowledgments

References

Aineto, D.; Celorrio, S. J.; and Onaindia, E. 2019. Learning action models with minimal observability. *Artificial Intelligence*, 275: 104–137.

Cresswell, S.; McCluskey, T. L.; and West, M. M. 2013. Acquiring planning domain models using *LOCM*. *Knowledge Engineering Review*, 28(2): 195–213.

Erol, K.; Hendler, J. A.; and Nau, D. S. 1994. HTN Planning: Complexity and Expressivity. In *Proc. of AAAI*, 1123–1128.

Esposito, F.; Semeraro, G.; Fanizzi, N.; and Ferilli, S. 2000. Multistrategy Theory Revision: Induction and Abduction in INTHELEX. *Machine Learning*, 38(1-2): 133–156.

Garland, A.; and Lesh, N. 2003. Learning hierarchical task models by demonstration. *MERL*.

Grand, M.; Fiorino, H.; and Pellier, D. 2020a. AMLSI: A Novel and Accurate Action Model Learning Algorithm. In *Proc of KEPS workshop*.

Grand, M.; Fiorino, H.; and Pellier, D. 2020b. Retro-engineering state machines into PDDL domains. In *Proc of ICTAI*, 1186–1193.

Hogg, C.; Kuter, U.; and Munoz-Avila, H. 2009. Learning hierarchical task networks for nondeterministic planning domains. In *Proc of IJCAI*.

Hogg, C.; Kuter, U.; and Munoz-Avila, H. 2010. Learning methods to generate good plans: Integrating htn learning and reinforcement learning. In *Proc of AAAI*, volume 24.

Hogg, C.; Munoz-Avila, H.; and Kuter, U. 2008. HTN-MAKER: Learning HTNs with Minimal Additional Knowledge Engineering Required. In *Proc of AAAI*, 950–956.

Höller, D. 2021. Translating totally ordered HTN planning problems to classical planning problems using regular approximation of context-free languages. In *Proc of ICAPS*, volume 31, 159–167.

Höller, D.; Behnke, G.; Bercher, P.; and Biundo, S. 2014. Language Classification of Hierarchical Planning Problems. In *Prof of ECAI*, 447–452.

Höller, D.; Behnke, G.; Bercher, P.; and Biundo, S. 2016. Assessing the Expressivity of Planning Formalisms through the Comparison to Formal Languages. In *Proc of ICAPS*, 158–165.

Höller, D.; Behnke, G.; Bercher, P.; Biundo, S.; Fiorino, H.; Pellier, D.; and Alford, R. 2019. Hierarchical Planning in the IPC. *CoRR*, abs/1909.04405.

Höller, D.; Behnke, G.; Bercher, P.; Biundo, S.; Fiorino, H.; Pellier, D.; and Alford, R. 2020. HDDL: An Extension to PDDL for Expressing Hierarchical Planning Problems. In *Proc of AAAI*, 9883–9891. AAAI Press.

Howey, R.; and Long, D. 2003. VAL's progress: The automatic validation tool for PDDL2. 1 used in the international planning competition. In *Proc of the International Workshop on the International Planning Competition*, 28–37.

Ilghami, O.; Nau, D. S.; and Muñoz-Avila, H. 2006. Learning to Do HTN Planning. In *Proc of ICAPS*, 390–393.

Ilghami, O.; Nau, D. S.; Muñoz-Avila, H.; and Aha, D. W. 2002. CaMeL: Learning Method Preconditions for HTN Planning. In *Proc of ICAPS*, 131–142.

Jungnickel, D. 1999. The Greedy Algorithm. In *Graphs, Networks and Algorithms*, 129–153. Springer.

Karp, R. M. 1972. Reducibility Among Combinatorial Problems. In *Proc. of a symposium on the Complexity of Computer Computations*, 85–103.

Li, N.; Cushing, W.; Kambhampati, S.; and Yoon, S. 2014. Learning probabilistic hierarchical task networks as probabilistic context-free grammars to capture user preferences.

ACM Transactions on Intelligent Systems and Technology, 5(2): 1–32.

Mourão, K.; Zettlemoyer, L. S.; Petrick, R. P. A.; and Steedman, M. 2012. Learning STRIPS Operators from Noisy and Incomplete Observations. In *Proc of UAI*, 614–623.

Nargesian, F.; and Ghassem-Sani, G. 2008. LHTNDT: Learn HTN Method Preconditions using Decision Tree. In *Proc of ICINCO-ICSO*, 60–65.

Nau, D. S.; Au, T.; Ilghami, O.; Kuter, U.; Muñoz-Avila, H.; Murdock, J. W.; Wu, D.; and Yaman, F. 2005. Applications of SHOP and SHOP2. *IEEE Intelligent Systems*, 20(2): 34–41.

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

Pellier, D.; and Fiorino, H. 2018. PDDL4J: a planning domain description library for java. *Journal of Experimental & Theoretical Artificial Intelligence*, 30(1): 143–176.

Pellier, D.; and Fiorino, H. 2020. Totally and Partially Ordered Hierarchical Planners in PDDL4J Library. *CoRR*, abs/2011.13297.

Xiaoa, Z.; Wan, H.; Zhuoa, H. H.; Herzigb, A.; Perrusselc, L.; and Chena, P. 2019. Learning HTN Methods with Preference from HTN Planning Instances. *Proc of HPlan workshop*, 31.

Yang, Q.; Wu, K.; and Jiang, Y. 2007. Learning action models from plan examples using weighted MAX-SAT. *Artificial Intelligence*, 171(2-3): 107–143.

Zhuo, H. H.; Hu, D. H.; Hogg, C.; Yang, Q.; and Muñoz-Avila, H. 2009. Learning HTN Method Preconditions and Action Models from Partial Observations. In *Proc of IJCAI*, 1804–1810.

Zhuo, H. H.; Nguyen, T. A.; and Kambhampati, S. 2013. Refining Incomplete Planning Domain Models Through Plan Traces. In *Proc of IJCAI*, 2451–2458.