Learning Applicability Conditions in AI Planning from Partial Observations

Hankz Hankui Zhuo, Derek Hao Hu and Qiang Yang
Department of Computer Science and Engineering, Hong Kong University of Science and Technology, Hong Kong
{hankz,derekhh,qyang}@cse.ust.hk

Héctor Muñoz-Avila and Chad Hogg
Department of Computer Science & Engineering,
Lehigh University
Bethlehem, Pennsylvania 18015, USA
{hem4, cmh204}@lehigh.edu

Abstract

AI planning has become more and more important in many real-world domains such as military applications and intelligent scheduling. However, planning systems require complete specifications of domain models, which can be difficult to encode, even for domain experts. Thus, research on effective and efficient methods to construct domain models or applicability conditions for planning automatically has become a hot topic for researchers. In this paper, we review our previous work ARMS, which can learn the applicability conditions for planning under STRIPS representations. Moreover, we provide two extensions to our ARMS system, LAMP, which can learn complex action models in PDDL representations with quantifiers and logical implications, and HTN-Learner, which can simultaneously learn method preconditions and action models in hierarchical task network (HTN) models. Our experimental results show that the two proposed algorithms could effectively learn complex action models and HTN models, thus having the ability to effectively acquire applicability conditions and relationships between actions in AI planning.

1 Introduction

AI planning has become more and more important in many domains such as military applications and intelligent scheduling. However, planning systems require complete domain models as input, which are difficult to acquire, even for domain experts. Nowadays, new planning domain specification languages like PDDL[Fox and Long, 2003] and HTN [Erol *et al.*, 1994] make it more difficult to specify applicability conditions[1] manually for domain experts, since these new representations involve the usage of universal and existential quantifiers, or the relationship between tasks and subtasks. These complex relationships are difficult to encode even for human when facing a rather complicated domain.

[1]We use the term "applicability conditions" instead of the more standard term "action models" since the HTN models require the specification of method preconditions, as well as action models for each action.

Because of the difficulty of manually providing applicability conditions, some researchers have developed methods to learn action models from the complete state information given before and after an action in some example plan traces. For example, Shahaf, Chang and Amir [Shahaf *et al.*, 2006] have proposed an algorithm called *Simultaneous Learning and Filtering* (SLAF) to learn more expressive action models using consistency-based algorithms.

The success of these learning systems all rely on the assumption that fully intermediate states are available as input. However, there may be only partially observed state information available in real-world planning applications, and in some domains, we cannot always guarantee the state information we had observed is correct. We provide two real-world examples to support our motivation for learning applicability conditions from partial observations.

In many operating systems, batch commands consist of the name of each command along with some partial information about directory location, structure and content. If we view such an application as a planning domain, the batch command is a list of actions. We can easily get the action name and its parameters (defined as action schema onwards) of each action by reading the system manual. However, we cannot get the full intermediate state information between these commands by reading the batch command file alone. This corresponds to an example of plan traces with partial intermediate state information. In this domain, we don't have the specification of action models, but we can easily get a large number of batch commands, viewed as plan examples, together with partial state information.

Another application is activity recognition [2], which is very important in pervasive computing, machine learning and wireless sensor networks. Activity recognition aims to recognize the actions and goals of one or more agents from a series of observations on the agents' actions and the environmental conditions. One scenario is sensor-based activity recognition [Hu and Yang, 2008], where we could use the sensor readings in a pervasive environment to understand what activities are going on by collecting a large number of sensor reading sequences and perform activity recognition on these sequences to get the corresponding activity sequences. These activity sequences can act as input of action model learning algo-

[2]http://en.wikipedia.org/wiki/Activity_recognition

rithms. However, due to the instability and noisy nature of sensor readings, the activities recognized by activity recognition algorithms may have some errors. Thus, the state information in the activity sequences learned may be sometimes incorrect.

Our previous work in learning action models, called ARMS [Yang *et al.*, 2007b], learns STRIPS action models from plan traces without or only with partial intermediate state observations under a STRIPS representation. Such a model seems to satisfy the real-world requirement where state observation may be incomplete or incorrect. However, as we mentioned before, the development of models like PDDL, which has quantified formulas and logical implications, and HTN, which has task decomposition structures, has posed out new challenges for action model learning tasks. Algorithms that could learn more expressive applicability conditions are needed. In this paper, we would briefly describe two new algorithms for learning applicability conditions that could solve the aforementioned challenges.

The first algorithm, LAMP, aims to learn action models for domains that can be expressed using a PDDL-like representation, or more precisely, in terms of quantifiers and logical implications. LAMP aims to enhance the "expressibility" of action models we learn by supporting the usage of quantifiers and implications in the final output. For instance, if there are different *briefcases* in a *briefcase*[3] domain where briefcases have different priorities to be moved away from a starting place. We can model the action *move* in PDDL as follows. [4]

action:	move(?c1 - case ?l1 ?l2 - location)
pre:	(:and (forall ?c2 - case
	(imply (prior ?c2 ?c1)(not (at ?c2 ?l1))))
	(at ?c1 ?l1))
effect:	(:and (at ?c1 ?l2) (not (at ?c1 ?l1)))

That is, if we want to move the case *c1* from a location *l1* to *l2*, *c1* should be at *l1* and every case *c2* priors to *c1* should not be at *l1*. After the action *move*, *c1* will be at *l2* instead of at *l1*. Here, we need universal quantifiers as well as logical implications in the precondition part of the action to precisely represent this domain and to compress the action models in a compact form.

The second algorithm, HTN-Learner, aims to learn applicability conditions from given structural traces of HTN models. Besides learning action models which are needed to represent primitive tasks in HTNs, we can also learn the preconditions of HTN methods, thus providing some understandings on the relationship between tasks and subtasks. Such a perspective is different from LAMP as it is not focusing on enhancing the *expressibility* but understanding the *relationship* between *tasks* at different levels of abstraction.

In the following, we first briefly review the ARMS algorithm and then introduce LAMP and HTN-Learner, respectively. After that, we would provide some preliminary experimental results to demonstrate the effectiveness of this three algorithms and show that LAMP and HTN-Learner could indeed enhance the *expressiveness* or explore the *relationship* between tasks. Next, we discuss some related work on action model learning. Finally, we conclude this paper and discuss some possible directions for future work.

[3]http://www.informatik.uni-freiburg.de/∼koehler/ipp/pddl-domains.tar.gz

[4]Notice that a symbol with a prefix "?" indicates that the symbol is a variable; e.g. "?c1" suggests that "c1" is a variable that can take on certain constants as values.

2 Learning Applicability Conditions

2.1 Definitions

A planning domain is defined as $\Sigma = (S, A, \gamma)$, where S is the set of states, A is the set of action models, γ is the deterministic transition function $S \times A \rightarrow S$. Each action model in A is composed of three parts [Fikes and Nilsson, 1971]: an action name with zero or more arguments, a precondition list which is a list of formulas that must hold in a state for the action to be applicable, an add list and a delete list that are sets of atoms. A planning problem can be defined as $\mathcal{P} = (\Sigma, s_0, g)$, where s_0 is an initial state, and g is a goal state. A solution to a planning problem is an action sequence $(a_0, a_1, \ldots, a_n)$ called a plan, which makes a projection from s_0 to g. Each a_i is an *action schema* composed of an action name and zero or more arguments. Furthermore, a *plan trace* is defined as $T = (s_0, a_0, s_1, a_1, \ldots, s_n, a_n, g)$, where $s_1, ..., s_n$ are partial intermediate state observations that are observed. They are called "partial" because they are allowed to be empty. The problem of learning applicability conditions can be stated as follows: given a set of plan traces T, output a set of applicability conditions for such plan traces to be able to proceed.

2.2 Learning STRIPS Action Models (ARMS)

We first briefly review our previous work ARMS [Yang *et al.*, 2007b]. ARMS proceeds in two phases. In phase one of the algorithm, ARMS finds frequent action sets from plans that share a common set of parameters. In addition, ARMS finds some frequent predicate-action pairs with the help of the initial state and the goal state. These predicate-action pairs give us an initial guess on the preconditions, add lists and delete lists of actions in this subset. These action subsets and pairs are used to obtain a set of constraints that must hold in order to make the plans correct. In phase two, we transform the constraints extracted from the plans into a weighted SAT representation [Moskewicz *et al.*, 2001], solve it, and produce an action model from the solution of the SAT problem. The process iterates until all actions are modeled.

The algorithm starts by initializing a set of explained actions and a set of action schemata yet to be explained. Subsequently, it iteratively builds a weighted MAX-SAT representation and solve it. Each time a few more actions are explained, and are used to build new initial states. At any intermediate stage, the partially learned action schemata increases in size monotonically. ARMS terminates when all actions in the example plans are learned. As a result, the actions of all plans examples are learned in a left-to-right sweep if we assume the plans begin on the left and end on the right side, without skipping any incomplete actions in between. Interested readers please refer to [Yang *et al.*, 2007b] for a more detailed description of the ARMS algorithm.

2.3 Learning Complex Action Models (LAMP)

Compared to ARMS, in this work, our algorithm LAMP attempts to learn more complex action models that support the use of quantifiers and logic implications, defined in PDDL language for simplifying previous STRIPS representations and in some situations even increasing the expressiveness of the possible constructs. Similar to ARMS, the learning problem of LAMP can be stated as, given a set of plan traces T, LAMP outputs a set of complex action models.

Each complex action model can be described as a set of formulas, e.g., the action "move" given in Section 1 can be described by the following formulas:

ID	formulas
1	$\forall$?c2,i. ((move ?c1 ?l1 ?l2 i) $\wedge$ (prior ?c2 ?c1 i) $\rightarrow \neg$(at ?c2 ?l1 i))
2	$\forall$ i. ((move ?c1 ?l1 ?l2 i) $\rightarrow$ (at ?c1 ?l1 i))
3	$\forall$ i. ((move ?c1 ?l1 ?l2 i) $\rightarrow$ (at ?c1 ?l2 i+1))
4	$\forall$ i. ((move ?c1 ?l1 ?l2 i) $\rightarrow \neg$(at ?c1 ?l1 i+1))

where, (move ?c1 ?l1 ?l2 i) is an atom indicating that the action (move ?c1 ?l1 ?l2) is executed in step i of a plan trace when the atom is *true*, likewise for other atoms. The first formula describes the precondition "(imply (prior ?c2 ?c1)(not (at ?c2 ?l1)))" of action "move". Other formulas can be explained similarly. Thus, in order to learn complex action models, we generate all the possible formulas, which are called *candidate formulas*, to encode all the possible complex action models. Since such candidate formulas could be very large, which makes it difficult to process, we constrain a logic implication with the form of $p \rightarrow q$, where p and q can be any predicate which probably appears in a complex action model. In this way, the number of candidate formulas are generally small since the number of actions and predicates is small, e.g., there are only 3 actions and 3 predicates in the domain *briefcase*.

Generally, LAMP can be described in four steps. Firstly, we encode the input plan traces, including observed states and actions (represented as state transitions), into propositional formulas. Secondly, we generate candidate formulas, according to the predicate lists and specific correctness constraints. Thirdly, we build a Markov Logic Network (MLN) [Domingos *et al.*, 2006] by learning the corresponding weight of each formula to select the most likely subset from the set of candidate formulas. Finally, we convert this subset into the final action models.

2.4 Learning Action Models and Method Preconditions (HTN-learner)

A Hierarchical Task Network (HTN) planning problem is defined as a quadruplet (s_0, T, M, A), where s_0 is an initial state which is a conjunction of propositions, T is a list of tasks that need to be accomplished, M is a set of *methods*, which specify how a high-level task can be decomposed into a totally ordered set of lower-level subtasks, and A is a set of *actions*, which corresponds to the primitive subtasks that can be directly executed. A *solution* to an HTN problem (s_0, T, M, A) is a list of *decomposition trees*. In a decomposition tree, a leaf node is a fully instantiated action, and the set of actions can be directly executed from the initial state to accomplish the root

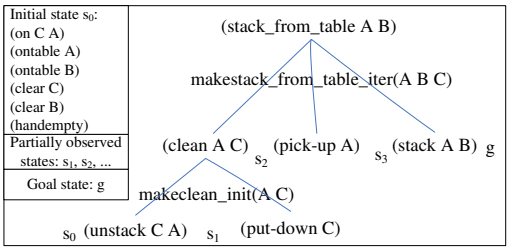

Figure 1: input: an example decomposition tree

level task. All intermediate level subtasks are also fully instantiated, and all preconditions of actions and preconditions of methods are satisfied. The roots of the trees correspond to the tasks in T.

Our learning problem can be described as follows: given as input a list of decomposition trees with partially observed states between the leaves of each decomposition tree, our learning algorithm outputs an HTN model including the *action models* and *method preconditions*. An example[5] of the input is shown in Figure 1. Figure 1 is an example of a decomposition tree with initial state or partially observed intermediate states (not shown in the figure) between leaves. The output of our algorithm HTN-Learner is the action models and method preconditions.

To learn action models and method preconditions, HTN-learner first builds the various constraints from the observed state information, including state constraints, decomposition constraints and action constraints, which are described as follows.

state constraint In a decomposition tree, if a predicate frequently appears before an action is executed, and its parameters are also parameters of the action, then the predicate is probably a precondition of the action. Likewise, if a predicate frequently appears before a method is applied, it is probably a precondition of the method; if a predicate frequently appears after an action is executed, it is probably an effect of the action. In other words, these frequent predicates provide useful information for our learning target. This information will be used in the form of *state constraints* in our learning process.

decomposition constraint *Decomposition constraints* is extracted from decomposition trees to encode the structure information. If a task T can be decomposed into n subtasks st_1, st_2, ..., st_n, we find that a subtask st_i often provides some preconditions of a method for subtask st_{i+1}, making that method applicable. As a result, this method can be applied to the next subtask st_{i+1}. Furthermore, we consider the constraint that the parameters of a precondition (or effect) should be included by the

[5] 'clean' is a task to move off all the blocks above '?x', and 'makeclean_init' is a method to be applied to 'clean'. 'stack_from_table' is a task to stack '?x' on '?y' when '?x' is on the table, and 'makestack_from_table_iter' is a method to be applied to 'stack_from_table'. 'pick-up', 'put-down', 'stack' and 'unstack' are four actions to pick up, put down, stack and un-stack a block.

parameters of the action or method the precondition (effect) belongs to.

action constraint To make sure that the learned action models are valid and reasonable and could reflect some characteristics of real-world action models, we need to further induce some constraints on different actions. These constraints are imposed on individual actions which can be divided into the following two types [Yang *et al.*, 2007b]: (1) An action is often required not to add a *fact* (instantiated atom) which already exists before the action is applied. (2) An action is also required that it should not delete a *fact* which does not exist before the action is applied.

For *state constraints* and *decomposition constraints*, we assign the weight of each constraint as the frequency of its appearance in decomposition trees. For *action constraints* we assign the weight of each constraint as the maximal weight of all the weights of *state constraints* and *decomposition constraints*, making action constraints to be satisfied maximally, which suggests that the learned action models will be close to real-world action models as much as possible.

With these constraints, all of which are associated with weights, `HTN-learner` builds a set of clauses and view it as a weighted maximum satisfiability problem, and then solves the problem by a weighted MAX-SAT solver[Borchers and Furman, 1998]. After that, `HTN-learner` converts the solution of the MAX-SAT problem to the HTN model including a set of action models and a set of HTN method preconditions that best explain the set of observed decomposition trees.

3 Experiment

To demonstrate the relationship of the learning results of these three learners, we extract from the domain *blocks world* [6] 30 plan traces for running `ARMS` and `LAMP` respectively, 30 decomposition trees for running `HTN-learner`. The results are given below.

By running `ARMS`, we can learn STRIPS action models of the domain *blocks world* as shown in the following.

(:action pick-up (?x - block)
:precondition (and**(clear ?x)**(ontable ?x)**(handempty)**)
:effect (and (not (ontable ?x)) **(not (clear ?x))***(clear ?x)*
(not (handempty))*(handempty)*(holding ?x)))
(:action put-down (?x - block)
:precondition (holding ?x) *(clear ?x)*
:effect (and **(not (holding ?x))(clear ?x)**
(handempty)*(not (clear ?x))***(ontable ?x)**))
(:action stack (?x - block ?y - block)
:precondition (and (holding ?x)**(clear ?y)***(ontable ?y)*)
:effect (and **(not (holding ?x))** (not (clear ?y))
(clear ?x)**(handempty)***(not(ontable ?y))*(on ?x ?y)))
(:action unstack (?x - block ?y - block)
:precondition (and **(on ?x ?y)** (clear ?x) (handempty))
:effect (and **(holding ?x) (clear ?y)** (not (clear ?x))
(not (handempty))*(on ?x ?y)* **(not (on ?x ?y))**))

Notice that, compared to the hand-written action models, parts in **bold** suggest that they should be added but not exist in the learned action models; parts in *italic* suggest that they should be deleted compared to the hand-written action models, but they are added in the learned action models. The same meaning is also used in the results of `LAMP` and `HTN-learner`.

By running `LAMP`, we can learn action models as show in the following.

(:action pick-up (?x - block)
:precondition (and (clear ?x)**(handempty)***(holding ?x)*)
:effect (and(not(handempty))(not(clear ?x))**(holding ?x)**
(when (ontable ?x)(not (ontable ?x)))
(forall (?y-block)(when(on ?x ?y)(clear ?y)))
(forall (?y-block)(when(on ?x ?y)(holding ?y)))
(forall(?y-block)(when(on ?x ?y)(not(on ?x ?y))))))
(:action put-down (?x - block)
:precondition (holding ?x) *(clear ?x) (handempty)*
:effect (and (not (holding ?x))**(clear ?x)**
(handempty) (ontable ?x)
(forall (?y-block)(when (not(clear ?y))(ontable ?x)))
(forall (?y-block)(when (clear ?y)(on ?x ?y)))))
(:action stack (?x - block ?y - block)
:precondition (and (holding ?x) *(clear ?y)(handempty)*)
:effect (and **(not (holding ?x))***(not (clear ?y))*(clear ?x)
(handempty) *(on ?x ?y)* (when (clear ?y)(on ?x ?y))
(when (ontable ?y)(on ?x ?y))
(when (ontable ?y)(not (clear ?y)))
(when (not(clear ?y))(ontable ?x))))
(:action unstack (?x - block ?y - block)
:precondition (and **(clear ?x)***(holding ?x)*(handempty))
:effect (and(not(handempty))**(not(clear ?x))***(ontable ?y)*
(clear ?x) **(holding ?x)**(when(on ?x ?y)(clear ?y))
(when(ontable ?y)(clear ?y))
(when(ontable ?x)(not(ontable ?x)))
(when(on ?x ?y)(not(on ?x ?y)))))

Compared to the learning result of `ARMS`, `LAMP` can learn action models with quantifiers (and conditional effects) and implications, e.g., "(forall (?y-block)(when(on ?x ?y)(clear ?y)))" in the action "pick-up". When these complex action models are learned, we observe that the action models "pick-up" and "unstack" are quite similar to each other. Actually, merging this two action as one action, named as "pick" or others, is reasonable in the domain *blocks world*, since we do not need to concern where a block is before we pick it up. That is to say, the similarities, specifically the complex action models learned, give us chances to merge actions together, reducing the action number as a result.

By running `HTN-learner` with decomposition trees as input, we can learn HTN method preconditions and action models. In the following, we only show one learned HTN method to give an intuitive idea about what `HTN-learner` learns because of lack of space.

(:method makestack_from_table_iter
:parameters (?x - block ?y - block ?z - block)
:task (stack_from_table ?x - block ?y - block)
:preconditions (and (ontable ?x) *(clear ?z) (holding ?z)*
(clear ?y) **(on ?z ?x)**)
:subtasks (and (clean ?x ?z) (pick-up ?x) (stack ?x ?y))

In the HTN method, "preconditions" are what we learned,

[6] http://www.cs.toronto.edu/aips2000/

while "task" and "subtasks" are assumed to be known beforehand in our algorithm HTN-learner.

4 Related Work

4.1 Action Model Learning

Recently, some researchers have proposed various methods to learn action models from plan traces automatically. The first one is to learn action models from plan traces with full intermediate state information [Benson, 1995; Wang, 1995; Schmill *et al.*, 2000; Pasula *et al.*, 2007]. [Schmill *et al.*, 2000] learns operators with approximate computation in relevant domains by assuming that the world is fully observable. [Wang, 1995] describes an approach to automatically learn planning operators by observing expert solution traces and refine the operators through practice in a learning-by-doing paradigm. [Benson, 1995] present methods by which an agent learns action models from its own experience and from its observation of a domain expert. It exploits the idea of concept induction in first-order predicate logic of inductive logic programming (ILP) [Muggleton and Raedt, 1994], which allows it to utilize ILP noise-handling techniques while learning without losing representational power. What they learn are STRIPS models[Fikes and Nilsson, 1971]. [Pasula *et al.*, 2007] show how to learn stochastic actions with no conditional effects. [Holmes and Jr., 2004] models synthetic items based on experience to construct action models. [Walsh and Littman, 2008] proposes an efficient algorithm for learning action schemas for describing Web services. Among these methods, one of their limitations is all the intermediate observations need to be known. However, in many real applications such as activity recognition from wireless sensor networks, biological applications of AI Planning, intelligent user interfaces and Web services [Ghallab *et al.*, 2004; Kuter *et al.*, 2005], sometimes we cannot obtain full intermediate state information.

SLAF [Amir, 2005; Shahaf and Amir, 2006] presents a tractable, exact solution for the problem of identifying actions' effects in partially observable STRIPS domains. It resembles version spaces and logical filtering and identifies all the models that are consistent with observations. It maintains and outputs a relational logical representation of all possible action-schema models after a sequence of executed actions and partial observations. To improve the performance, [Shahaf *et al.*, 2006] proposes an efficient algorithm to learn preconditions and effects of deterministic action models. In many real-world planning applications, however, existential quantifiers may appear in the effects of actions and universal quantifiers may appear in the preconditions of some actions, as shown in the examples of "*move*".

4.2 Markov Logic Networks (MLNs)

The Markov Logic Network (MLN) [Richardson and Domingos, 2006] is a powerful framework that combines probability and first-order logic with statistical learning. An advantage of MLNs over first order logic is in its ability to "soften" the constraints; e.g., when a world violates a formula in a knowledge base, it is less probable, but not impossible. Thus, each formula is associated with a *weight* to reflect how strong the constraint is. Weights can be learned using a variety of methods, e.g. convex optimization of the likelihood, iterative scaling and margin maximization. MLNs have been applied to several real world applications with great success. For instance, [Domingos, 2005] proposes to apply Markov logic to model real social networks, which evolve in time with multiple types of arcs and nodes and are affected by the actions of multiple players; [Poon and Domingos, 2007] proposes a joint approach to perform information extraction using Markov logic and existing algorithms, where segmentation of all records and entity resolution are performed together in a single integrated inference process.

4.3 HTN learning

We are focusing on a variant of HTN planning called Ordered Task Decomposition [Nau *et al.*, 2005], which is also the most common variant of HTN planning by far. In this variant the planning system generates a plan by decomposing tasks in the order they were generated into simpler and simpler subtasks until primitive tasks are reached that can be performed directly. Specifically, for each non-primitive task, the planner chooses an applicable method and instantiates it to decompose the task into subtasks. When the decomposition process reaches a primitive subtask, the planner accomplishes it by applying its corresponding action in the usual STRIPS fashion. The process stops when all non-primitive tasks are decomposed into primitive subtasks, and outputs an action sequence (i.e. a plan) as a solution.

[Ilghami *et al.*, 2005; Xu and Muñoz-Avila, 2005] propose eager and lazy learning algorithms respectively, to learn the preconditions of HTN methods. These systems require as input the hierarchical relationships between tasks, the action models, and a complete description of the intermediate states and learn the conditions under which a method may be used. Icarus uses means-end analysis to learn structure and preconditions of the input plans by assuming that a model of the tasks in the form of Horn clauses is given [Nejati *et al.*, 2006]. [Yang *et al.*, 2007a] presents a probabilistic model for unsupervised learning of HTN methods from action sequences. HTN-MAKER also learns structures albeit assuming that a model of the tasks is given in the form of preconditions and effects for the tasks [Hogg *et al.*, 2008]. All of these works have in common that they assume complete intermediate state information to be given together with the input traces.

5 Conclusion

In this paper, we have given an overview on several novel approaches to learn applicability conditions in AI Planning from partial observations, including STRIPS action models, complex action models with quantifiers and logical implications, as well as HTN models including action models and method preconditions, from a set of observed plan traces or decomposition trees where we can support partially observable intermediate states. We list several possible directions which we could follow for our future work. Our current LAMP algorithm enumerates all possible preconditions and effects according to our specific correctness constraints. In the future, we wish to add some form of domain knowledge to further

filter out some "impossible" candidate formulas beforehand thereby making the algorithm much more efficient. Another direction we want to explore is to extend the action model learning algorithm to more elaborate action representation languages that explicitly represent resources and functions. We will also apply our algorithms to more challenging tasks in real world planning applications.

Acknowledgment

We thank the support of Hong Kong CERG Grant HKUST 621307, NEC China Lab and the National Science Foundation Grant No. NSF 0642882.

References

[Amir, 2005] E. Amir. Learning partially observable deterministic action models. In *Proceedings of IJCAI'05*, pages 1433–1439, 2005.

[Benson, 1995] Scott Benson. Inductive learning of reactive action models. In *Proceedings of ICML'95*, 1995.

[Borchers and Furman, 1998] B. Borchers and J. Furman. A two-phase exact algorithm for MAX-SAT and weighted MAX-SAT problems. *J. Comb. Optim.*, 2(4), 1998.

[Domingos *et al.*, 2006] P. Domingos, S. Kok, H. Poon, M. Richardson, and P. Singla. Unifying logical and statistical ai. In *Proceedings of AAAI'06*, 2006.

[Domingos, 2005] P. Domingos. Mining social networks for viral marketing. *IEEE Intelligent Systems*, 20(1), 2005.

[Erol *et al.*, 1994] Kutluhan Erol, James A. Hendler, and Dana S. Nau. Umcp: A sound and complete procedure for hierarchical task-network planning. In *Proceedings of AIPS'94*, pages 249–254, 1994.

[Fikes and Nilsson, 1971] R. Fikes and N. J. Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence Journal*, pages 189–208, 1971.

[Fox and Long, 2003] Maria Fox and Derek Long. Pddl2.1: An extension to pddl for expressing temporal planning domains. *J. Artif. Intell. Res. (JAIR)*, 20:61–124, 2003.

[Ghallab *et al.*, 2004] Mark Ghallab, Dana Nau, and Paolo Traverso. *Automated Planning: Theory and Practice*. Morgan Kaufmann Publishers, 2004.

[Hogg *et al.*, 2008] C. Hogg, H. Muñoz-Avila, and U. Kuter. HTN-MAKER: Learning HTNs with minimal additional knowledge engineering required. In *Proceedings of AAAI'08*, pages 950–956, 2008.

[Holmes and Jr., 2004] M. P. Holmes and C. L. I. Jr. Schema learning: Experience-based construction of predictive action models. In *Advances in NIPS'04*, 2004.

[Hu and Yang, 2008] Derek Hao Hu and Qiang Yang. CIGAR: Concurrent and interleaving goal and activity recognition. In *Proceedings of AAAI'08*, 2008.

[Ilghami *et al.*, 2005] O. Ilghami, H. Muñoz-Avila, D. S. Nau, and D. W. Aha. Learning approximate preconditions for methods in hierarchical plans. In *Proceedings of ICML'05*, 2005.

[Kuter *et al.*, 2005] Ugur Kuter, Evren Sirin, Bijan Parsia, Dana Nau, and James Hendler. Information gathering during planning for web service composition. *Journal of Web Semantics (JWS)*, pages 183–205, 2005.

[Moskewicz *et al.*, 2001] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient sat solver. In *Proceedings of DAC'01*, 2001.

[Muggleton and Raedt, 1994] Stephen Muggleton and Luc De Raedt. Inductive logic programming: Theory and methods. *Journal of Logic Programming*, 1994.

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

[Nejati *et al.*, 2006] N. Nejati, P. Langley, and T. Konik. Learning hierarchical task networks by obervation. In *Proceedings of ICML'06*, pages 665–672, 2006.

[Pasula *et al.*, 2007] H. M. Pasula, L. S. Zettlemoyer, and L. P. Kaelbling. Learning symbolic models of stochastic domains. *Journal of Artificial Intelligence Research*, 2007.

[Poon and Domingos, 2007] H. Poon and P. Domingos. Joint inference in information extraction. In *Proceedings of AAAI'07*, pages 913–918, 2007.

[Richardson and Domingos, 2006] Matthew Richardson and Pedro Domingos. Markov logic networks. *Machine Learning*, 62(1-2):107–136, 2006.

[Schmill *et al.*, 2000] M. D. Schmill, T. Oates, and P. R. Cohen. Learning planning operators in real-world, partially observable environments. In *Proceedings of AIPS'00*, 2000.

[Shahaf and Amir, 2006] Dafna Shahaf and Eyal Amir. Learning partially observable action schemas. In *Proceedings of AAAI'06*, pages 913–919, 2006.

[Shahaf *et al.*, 2006] Dafna Shahaf, Allen Chang, and Eyal Amir. Learning partially observable action models: Efficient algorithms. In *Proceedings of AAAI'06*, 2006.

[Walsh and Littman, 2008] Thomas J. Walsh and Michael L. Littman. Efficient learning of action schemas and web-service descriptions. In *Proceedings of AAAI'08*, 2008.

[Wang, 1995] Xuemei Wang. Learning by observation and practice: An incremental approach for planning operator acquisition. In *Proceedings of ICML'95*, 1995.

[Xu and Muñoz-Avila, 2005] K. Xu and H. Muñoz-Avila. A domain-independent system for case-based task decomposition without domain theories. In *Proceedings of AAAI'05*, 2005.

[Yang *et al.*, 2007a] Qiang Yang, Rong Pan, and Sinno Jialin Pan. Learning recursive htn-method structures for planning. In *Proceedings of the ICAPS-07 Workshop on AI Planning and Learning*, 2007.

[Yang *et al.*, 2007b] Qiang Yang, Kangheng Wu, and Yunfei Jiang. Learning action models from plan examples using weighted MAX-SAT. *Artificial Intelligence Journal*, 171:107–143, February 2007.