Learning HTN Method Preconditions and Action Models from Partial Observations

Hankz Hankui Zhuo[a], Derek Hao Hu[a], Chad Hogg[b], Qiang Yang[a] and Hector Munoz-Avila[b]

[a] Dept of Computer Science and Engineering
Hong Kong University of Science and Technology
Clear Water Bay, Kowloon, Hong Kong
{hankz, derekhh, qyang}@cse.ust.hk

[b] Dept of Computer Science & Engineering
Lehigh University
Bethlehem, PA, USA
{cmh204, munoz}@cse.lehigh.edu

Abstract

To apply hierarchical task network (HTN) planning to real-world planning problems, one needs to encode the HTN schemata and action models beforehand. However, acquiring such domain knowledge is difficult and time-consuming because the HTN domain definition involves a significant knowledge-engineering effort. A system that can learn the HTN planning domain knowledge automatically would save time and allow HTN planning to be used in domains where such knowledge-engineering effort is not feasible. In this paper, we present a formal framework and algorithms to acquire HTN planning domain knowledge, by learning the preconditions and effects of actions and preconditions of methods. Our algorithm, HTN-learner, first builds constraints from given observed *decomposition trees* to build action models and method preconditions. It then solves these constraints using a weighted MAX-SAT solver. The solution can be converted to action models and method preconditions. Unlike prior work on HTN learning, we do not depend on complete action models or state information. We test the algorithm on several domains, and show that our HTN-learner algorithm is both effective and efficient.

1 Introduction

In many real-world planning applications, HTN planning systems have shown their advantages in effectively using domain knowledge to solve planning problems [Nau *et al.*, 2005]. However, for HTN planning to be applicable, domain experts must first encode the HTN schemata and action models beforehand. Acquiring such domain knowledge is difficult and time-consuming since the HTN domain definition involves a significant knowledge-engineering effort. Thus, it is an important problem to be able to develop learning algorithms to help acquire the domain knowledge for HTN planning.

The problem of learning some interesting aspects of HTN domain knowledge has captured the attention of many researchers. In particular, learning the applicability conditions from given structural traces of HTNs has been the subject of recurrent research interest. In the works of [Ilghami *et al.*, 2005], an instance of this problem has been studied, albeit under the assumption that the preconditions and effects of actions in the domain are fully specified and there is complete observability of the states of the world. In [Xu and Muñoz-Avila, 2005] another instance of the problem is studied under the assumption that an ontology indicating relations between the objects is given. In the works of [Nejati *et al.*, 2006; Reddy and Tadepalli, 1997], approaches are developed to learn the hierarchical structure that relate the tasks and subtasks. Existing work on learning hierarchies elicits a hierarchy from a collection of plans and from a given action model. In [Hogg *et al.*, 2008], the HTN-MAKER algorithm learns the decomposition methods for Hierarchical Task Networks. However, so far all of these studies assume the complete observability of the intermediate states of the plans and a complete action model being given. In real world situations, we note that such learning problems are even more difficult since the observed plan traces we use as training data may contain incomplete and inconsistent information.

Table 1: Comparison of different HTN learning algorithms

compared algorithms	action models	partial observab.	method preconditions
HTN-learner$^\flat$	$\surd$	$\surd$	$\surd$
HTN-MAKER$^\natural$	$\times$	$\times$	$\surd$
CaMeL, DInCAD$^\sharp$	$\times$	$\times$	$\surd$
Icarus, XLearn†	$\times$	$\times$	$\surd$

$\flat$: this paper.
$\natural$: [Hogg *et al.*, 2008].
$\sharp$: [Ilghami *et al.*, 2005; Xu and Muñoz-Avila, 2005].
$\dagger$: [Nejati *et al.*, 2006; Reddy and Tadepalli, 1997].

We present a new algorithm which we call HTN-learner that learns the HTN method preconditions and the preconditions and effects of actions in HTN planning under partial observability. Table 1 shows the major difference between our proposed learning problem and those of the past works. Each column denotes whether the corresponding algorithm could learn the action models or the method preconditions, and whether it supports partial observability of the plan states. As input to HTN-learner, we assume that we have the decomposition structure in the form of task decomposition trees whose leaves are all primitive actions, which we explain in

detail in Section 3. These trees are readily available in some real-world domains such as process planning, where human planners enter so-called Work-Breakdown Structures indicating the activities that must be accomplished for a project. These Work-Breakdown Structures can be mapped to hierarchical task networks [Xu and Muñoz-Avila, 2004]. The main problem for the automated reuse of this knowledge is that, whereas the instances of the structural relations are readily available for such domains, their applicability conditions are neither given nor expected to be given by the human planner.

We present a novel framework to acquire domain knowledge, including learning action models and method preconditions. Our algorithm runs in two steps. First, we build three kinds of constraints to encode the features of action models and method preconditions from the observed decomposition trees. Second, we solve these constraints using a weighted MAX-SAT solver [Borchers and Furman, 1998], and convert the result to action models and method preconditions.

In the following, we first present the related work in Section 2, and then define our learning problem in Section 3. After that, we give the detailed description of our main learning algorithm in Section 4. Finally, we give experimental results and discussion to show our algorithm is effective and efficient.

2 Related Work

2.1 HTN Planning

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.

2.2 Learning Action Models

ARMS (action-relation modeling system) [Yang *et al.*, 2007] presents a framework for automatically discovering STRIPS [Fikes and Nilsson, 1971] action models from a set of successfully observed plans. It gathers knowledge on the statistical distribution of frequent sets of actions in the example plans without assuming complete knowledge of states in the middle of observed plans. In knowledge acquisition for planning model learning, a computer system interacts with a human expert to generate the needed action models [Blythe *et al.*, 2001; McCluskey *et al.*, 2003], where the states just before or after each action are assumed to be known. [Amir, 2005] presented a tractable and exact technique for learning action models known as Simultaneous Learning and Filtering, where the state observations are needed for learning. While these systems can learn action models of various forms, they are mainly designed for learning for non-hierarchical planning algorithms.

2.3 HTN Learning

[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]. HTN-MAKER also learns structures albeit assuming that a model of the tasks is given in the form of preconditons and effects for the tasks [Hogg *et al.*, 2008].

3 Problem Definition

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.

In this definition, each task has a task name with zero or more arguments, each of which is either a variable symbol or a constant symbol. A method is defined as $(m, t, \text{PRE}, \text{SUB})$, where m is a unique method name with zero or more arguments, t is the head task the method decomposes, PRE are the preconditions of the method, and SUB is a list of subtasks into which the head task may be decomposed. The arguments of m consist of the arguments of the head task, the arguments of each of the subtasks, and all terms used in the preconditions. Each of the subtasks may be primitive, in which case they correspond to an action schema, or non-primitive, in which case they must be further decomposed. Each method precondition is a literal, and the set of method preconditions must be satisfied before the method can be applied. An *action model* a is defined as $a = (o, \text{PRE}, \text{ADD}, \text{DEL})$, where o is an action schema composed of an action name and zero or more arguments, PRE is a *precondition list*, ADD is an *add list* and DEL is a *delete list* [Fikes and Nilsson, 1971].

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 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 defined as: 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[1] of the input is shown

[1] 'clean' is a task to move off all the blocks above '?x',

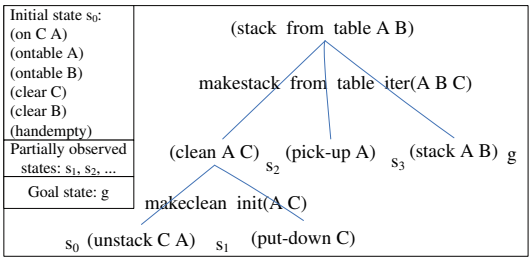

Figure 1: input: an example decomposition tree

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 is the action models and method preconditions.

4 Algorithm Description

We first present an overview of the algorithm in Section 4.1, and then provide the detailed description of each step in Sections 4.2-4.6.

4.1 Algorithm Framework

Our algorithm takes a set of decomposition trees as input, and produces preconditions and effects of actions as well as method preconditions as output. To reach this aim, it first builds the various constraints from the observed state information, including state constraints, decomposition constraints and action constraints. Based on these constraints, it will build a set of clauses and view it as a weighted maximum satisfiability problem, which is solved by a weighted MAX-SAT solver[Borchers and Furman, 1998]. The solution to this MAX-SAT problem is the HTN model including the set of action models and HTN method preconditions that best explains the set of observed decomposition trees. An overview of the algorithm is shown in Algorithm 1.

Algorithm 1 Algorithm overview of HTN-learner

Input: DTR: A set of decomposition trees with partially observed states between leaves.
Output: The HTN model H;

1: Extract the HTN schemata;
2: Build state constraints SC;
3: Build decomposition constraints DC;
4: Build action constraints AC;
5: Solve constraints using weighted MAX-SAT, and convert the result to the HTN model H;
6: **return** H;

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.

4.2 Step 1: Extracting the HTN Schemata

In this step, we extract the HTN schemata, including a predicate list, an action-schema list and a method-structure list. We use a straightforward process to do this step. Firstly, we scan all the decomposition trees and substitute all the *objects* with their corresponding variables, each of which is constrained by a *type*. Secondly, we collect (1) all the different predicates as a predicate list, each of which has its own arguments that are variables; (2) all the different action-schemas as an action-schema list, each of which is composed of an action name and zero or more arguments that are variables; (3) all the different decompositions, each of which is composed of a task and its corresponding subtasks, as a method-structure list. For example, from the decomposition tree in Figure 1, we can extract (1) a predicate list: {(on ?x-block ?y-block), (ontable ?x-block), (clear ?x-block)}; (2) an action-schema list: {(pick-up ?x-block),(stack ?x-block ?y-block), (unstack ?x-block ?y-block), (put-down ?x-block)}; (3) a method-structure list: {⟨(makestack_from_table_iter ?x-block ?y-block ?z-block), (*stack_from_table* ?x ?y) ((*clean* ?x ?z)(*pick-up* ?x)(*stack* ?x ?y))⟩; ⟨(makeclean_init ?x-block ?y-block), (*clean* ?x ?y)((*unstack* ?y ?x)(*put-down* ?y))⟩}, where a method-structure is described as "⟨ method-name, task (subtask$_1$ ... subtask$_k$)⟩".

4.3 Step 2: Building State Constraints

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 likely to be a precondition of the action. Likewise, if a predicate frequently appears before a method is applied, it is likely to be a precondition of the method; if a predicate frequently appears after an action is executed, it is likely to be an effect of the action. This information will be encoded in the form of constraints in our learning process. Since these constraints are built from the relations between states and actions, or states and methods, we call these constraints *state constraints*. The following is the process of building state constraints SC (PARA(p) denotes the set of parameters of p.):

(1). By scanning all the decomposition trees, for each predicate p in the state where an action a is executed and PARA(p) $\subseteq$ PARA(a), we build a constraint $p \in$ PRE(a), the set of which is denoted as SC$_{pa}$, which indicates the possible candidates of predicates that might be a precondition of action a.

(2). For each predicate p in the state after an action a is executed and PARA(p) $\subseteq$ PARA(a), we build a constraint $p \in$ ADD(a), the set of which is denoted as SC$_{ap}$. SC$_{ap}$ indicates the possible candidates of predicates that might be an effect of action a.

(3). For each predicate p in the state where a method m is applied, we build a constraint $p \in$ PRE(m), the set of which is denoted as SC$_{pm}$.

As a result, we get three kinds of constraints SC$_{pa}$, SC$_{ap}$ and SC$_{pm}$, which together form the state constraints SC. With SC, we build weighted state constraints WSC with the *calWeight*(SC) procedure that combines the instantiated constraints in SC into their corresponding variable-form constraints, and assigns weights to these variable-form con-

straints. The procedure of *calWeight*(SC) can be described by the following steps: (1) replace all the instantiated arguments in SC with their corresponding variables, denoted the results as C′; (2) calculate the output WSC of the procedure by WSC $= \{\langle w, c\rangle | c \in C' \wedge w = \text{numberOf}(c, C')\}$, where numberOf$(c, C')$ returns the number of c's appearances in C'.

4.4 Step 3: Building Decomposition Constraints

In this step, we build decomposition constraints to encode the structure information provided by decomposition trees. 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 parameters of the action or method the precondition (effect) belongs to. As a result, the decomposition constraints DC can be built by the procedure as shown in Algorithm 2.

Algorithm 2 Build decomposition constraints: *buildDecmpConstr(*DTR*)*

Input: A set of decomposition trees with partially observed states DTR;
Output: Decomposition constraints DC;

```
DC = ∅;
for each decomposition tree dtr ∈ DTR do
    for each two subtasks st_i and st_j in dtr do
        if i < j then
            for k = 1 to n_i do
                PRS = PARA(a_ik) ∩ PARA(m_j);
                generate a set of predicates GP using PRS;
                generate a constraint c and add it to DC;
            end for
        end if
    end for
end for
return DC;
```

In the fourth step of Algorithm 2, we consider two tasks st_i and st_j that have the same parent and st_i occurs earlier than st_j. In the fifth step, n_i is the number of actions to accomplish the subtask st_i, which is denoted as a_{i1}, a_{i2}, …, a_{in_i}. In the sixth step, m_j is the method which is applied to the subtask st_j. In the seventh step, GP is generated by GP $= \{p|\text{PARA}(p) \subseteq \text{PRS}\}$. In the last step, the generated constraint c is: $p \in \text{GP} \rightarrow (p \in \text{ADD}(a_{ik}) \wedge p \in \text{PRE}(m_j))$ With DC, we build weighted decomposition constraints WDC by setting: WDC = *calWeight*(DC), which is similar to the procedure of calculating WSC. WDC can be solved directly by a weighted MAX-SAT solver.

4.5 Step 4: Building Action Constraints

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.*, 2007]:
(1). An action may not add a *fact* (instantiated atom) which already exists before the action is applied. This constraint can be encoded as $p \in \text{ADD}(a) \Rightarrow p \notin \text{PRE}(a)$, where p is an atom, ADD(a) is a set of added effects of the action a, and PRE(a) is a set of preconditions of a.
(2). An action may not delete a *fact* which does not exist before the action is applied. This constraint can be encoded as $p \in \text{DEL}(a) \Rightarrow p \in \text{PRE}(a)$, where DEL$(a)$ is a set of delete effects of a.

These constraints are placed to ensure the learned action models are succinct, and most existing planning domains satisfy them. Nevertheless, our learning algorithm works perfectly without them.

These constraints compose the action constraints AC. We denote as w_{max} the maximal weight of all the constraints in WSC and WDC, and assign w_{max} as the weight of all constraints in AC. In this way, the weights of constraints in AC are not less than the ones in WSC or WDC, which suggests that the action constraints AC should be satisfied in many real applications, compared to the other two kinds of constraints.

4.6 Step 5: Solving the Constraints

By using Steps 2-4, three kinds of weighted constraints are built to encode the information of action models and method preconditions. Before the constraints can be solved by a weighted MAX-SAT solver, the relative importance of these *three kinds* of constraints must be determined. To do this, we introduce three new parameters β_i $(1 < i < 3)$ to control weights of each kind of constraints by: $\frac{\beta_i}{1-\beta_i} w_i$, where $0 \leq \beta_i < 1$, $1 \leq i \leq 3$, and w_i is a weight of the ith kind of constraint. Notice that, by using $\frac{\beta_i}{1-\beta_i}$, we can easily adjust the weight from 0 to ∞ by simply adjusting β_i from 0 to 1. The weight w_i will be replaced by $\frac{\beta_i}{1-\beta_i} w_i$, and the resulting weighted constraints are solved by a weighted MAX-SAT solver. As a result, a *true* or *false* assignment will be outputted to maximally express the weighted constraints. According to the assignment, the HTN model can be acquired directly. For instance, if "$p \in$ ADD(a)" is assigned *true* in the result of the solver, then p will be converted into an effect of the action a in the HTN model.

5 Experiment

5.1 Datasets

In this section, we performed experiments to evaluate our algorithm. In the experiment, we use the HTN domains called *htn-blocks*, *htn-depots* and *htn-driverlog* for training and testing, which are created as HTN domains based on the domains *blocks world* from IPC-2 [2] and *depots, driverlog* from IPC-3 [3] respectively. We generate 200 decomposition trees from each domain for training the HTN model, and compare the result to its corresponding handwritten HTN model.

[2] http://www.cs.toronto.edu/aips2000/
[3] http://planning.cis.strath.ac.uk/competition/

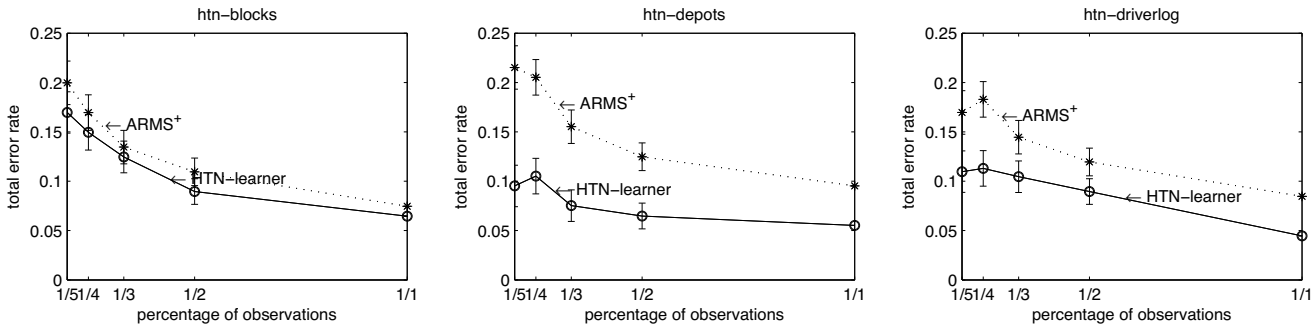

Figure 2: the total error with respect to the percentage of observations

5.2 Evaluation Metric

To evaluate our algorithm, we define two kinds of errors: *soundness error* and *completeness error*. If a precondition (or an effect) does not exist in the learned HTN model, while it did in the original model, then we call this situation a *soundness error* because this error will likely result in situations where the learned method (or action) is applied while the original one was not. If a precondition (or an effect of an action) exists in the learned model, while it is not in the original one, then we call this situation a *completeness error* because this error will likely result in a learned method (or action) not being applied in situations where the original one is applicable. We denote the soundness error rate of an HTN model as E_s, the completeness error rate as E_c, and the total error rate of an HTN model as E_t, where $E_t = E_s + E_c$. Then we calculate E_s and E_c as follows: $E_s = \sum_a E_s(a) = \sum_a \frac{\text{soundness errors of } a}{\text{all possible conditions of } a}$, and $E_c = \sum_a E_c(a) = \sum_a \frac{\text{completeness errors of } a}{\text{all possible conditions of } a}$, where a can be an action or a method.

5.3 Experimental Results

An alternative to HTN-learner also solving this problem would be to learn the action models with ARMS [Yang *et al.*, 2007] and separately learn the method preconditions with an existing algorithm such as CaMeL [Ilghami *et al.*, 2005]. To determine the importance of learning the action models and method preconditions simultaneously, we ran an experiment comparing HTN-learner against a hybrid system, that we call it ARMS$^+$, which first uses ARMS to learn the action models and then uses the method-based constraints to learn the method preconditions.

The results of this experiment are shown in Figure 2. We varied the percentage of intermediate states provided from 1/5 to 1 and set each β_i to 0.5. In all cases, the error rate of HTN-learner was less than that of ARMS$^+$. This was expected, because constraints about the action models may provide information that may be exploited to find more accurate method preconditions, and vice versa. The differences are more significant on the more complex domains such as htn-depots, where combining the structural and action models provides additional useful constraints that would not be generated from the two separately. An increase in the number of intermediate states that are specified generally increases the accuracy of both systems.

We now analyze the relative importance of the three types of constraints, to show that all of them are needed to help improve the learning result. Therefore, with respect to different β_i, we train an HTN model by setting the percentage of observations as 1/4, and calculate its total errors E_t. From Figure 3(a), we can see that E_t first decreases when β_1 increases, which means the state constraints should be more important to improve the correctness. Then, E_t increases and keeps stable at a constant value, that is because when β_1 reaches a high enough value, the importance of the two other kinds of constraints is reduced, and it plays a negative effect on our learning accuracy. From Figure 3(b), we can see that, E_t decreases at first but increases quickly when β_2 increases. After β_2 reaches 0.5, E_t becomes unstable with respect to different domains. This shows that, since the importance of other constraints decreases when β_2 is high enough, exploiting the information of decomposition constraints is not good enough to extract useful rules to learn action models. The curve in Figure 3(c) is similar to Figure 3(a), except that it is sharper than the one in Figure 3(a) before β_3 reaches 0.5, which suggests that the information from action constraints is stronger than that from state constraints before 0.5.

To test the running time of HTN-learner, we set the percentage of observations as 1/4 and test HTN-learner with respect to different number of decomposition trees. The testing result is shown in Table 2. The running time of our algorithm increases polynomially with the size of the input. To verify our claim, we use the relationship between the size of input and the CPU running time to estimate a function that could best fit these points. We've found that we are able to fit the performance curve with a polynomial of order 2 or order 3. We provide the polynomial for fitting *htn-driverlog*, which is $0.0110x^2 - 0.0410x - 1.2000$. We also observed that the total error rate decreases when the size of the input increase, and it remains below 0.12 in all cases where there are more than 140 decomposition trees available.

6 Conclusion

In this paper, we have presented a novel algorithm HTN-learner to learn the action models and method preconditions of an HTN model. Given a set of partially observed decomposition trees, HTN-learner builds a set of state, decomposition, and action weighted constraints, and solves them with a

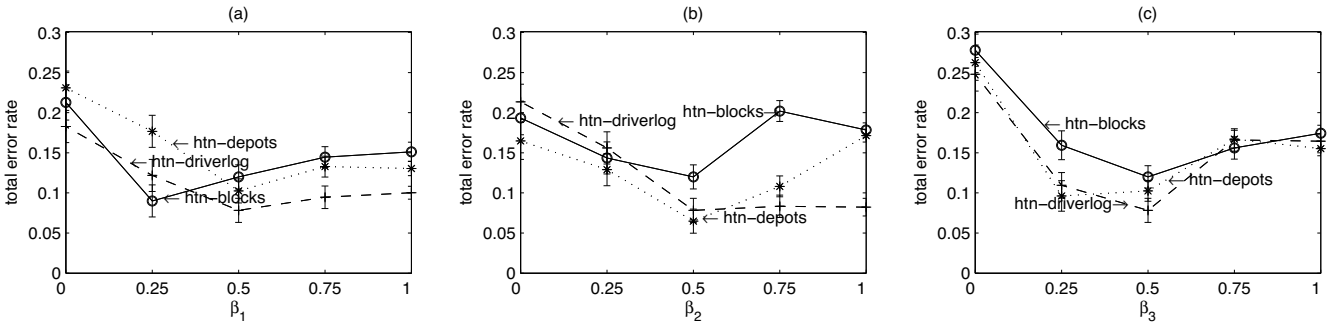

Figure 3: the total error with respect to (a) β_1, (b) β_2 or (c) β_3

number	htn-driverlog	htn-blocks	htn-depots
20	3	5	8
40	8	13	18
60	34	43	63
80	65	98	98
100	102	73	132
120	91	112	121
140	165	146	189
160	203	175	188
180	264	198	240
200	283	234	323

Table 2: column 1 is the number of decomposition trees, columns 2-4 are cpu times (seconds)

MAX-SAT solver. The solution obtained is the HTN model that best explains the observed decomposition trees.

We observed the following conclusions from our empirical evaluation in 3 benchmark domains: (1) Simultaneously solving the constraints reduces error in the learned HTN-model compared to first learning the action model and then learning the method's preconditions, (2) By imposing the three types of constraints, we can effectively reduce the hypothesis space of possible action models and method preconditions while using the input plan traces as an empirical basis, and (3) The running time of our algorithm increases polynomially and the error rate decreases with the size of the input. Our methods can reduce human efforts in acquiring HTN models and scale up planning in real-world applications.

Acknowledgment

We thank the support of Hong Kong CERG Grant HKUST 621307, NEC China Lab, and the National Science Foundation (NSF 0642882).

References

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

[Blythe *et al.*, 2001] J. Blythe, J. Kim, S. Ramachandran, and Y. Gil. An integrated environment for knowledge acquisition. In *Proceedings of IUI*, pages 13–20, 2001.

[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.

[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.

[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*, pages 950–956, 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*, pages 337–344, 2005.

[McCluskey *et al.*, 2003] T. L. McCluskey, D. Liu, and R. M. Simpson. GIPO II: HTN planning in a tool-supported knowledge engineering environment. In *Proceedings of ICAPS*, pages 92–101, 2003.

[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*, pages 665–672, 2006.

[Reddy and Tadepalli, 1997] C. Reddy and P. Tadepalli. Learning goal-decomposition rules using exercises. In *Proceedings of ICML*, pages 278–286, 1997.

[Xu and Muñoz-Avila, 2004] K. Xu and H. Muñoz-Avila. CaBMA: Case-based project management assistant. In *Proceedings of IAAI*, pages 931–936, 2004.

[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*, pages 234–240, 2005.

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