Learning Hierarchical Task Network Domains from Partially Observed Plan Traces

Hankz Hankui Zhuo[a], Héctor Muñoz-Avila[b], Qiang Yang[c,*]

[a]*Department of Computer Science, Sun Yat-sen University, Guangzhou, China*
[b]*Department of Computer Science and Engineering, Lehigh University, Bethlehem, PA, USA*
[c]*Department of Computer Science and Engineering, Hong Kong University of Science and Technology, Clearwater Bay, Kowloon, Hong Kong*

Abstract

Hierarchical Task Network (HTN) planning is an effective yet knowledge intensive problem-solving technique. It requires humans to encode knowledge in the form of *methods* and *action models*. Methods describe how to decompose tasks into subtasks and the preconditions under which those methods are applicable whereas action models describe how actions change the world. Encoding such knowledge is a difficult and time-consuming process, even for domain experts. In this paper, we propose a new learning algorithm, called `HTNLearn`, to help acquire HTN methods and action models. `HTNLearn` receives as input a collection of plan traces with partially annotated intermediate state information, and a set of annotated tasks that specify the conditions before and after the tasks' completion. In addition, plan traces are annotated with potentially empty *partial* decomposition trees that record the processes of decomposing tasks to subtasks. `HTNLearn` outputs are a collection of methods and action models. `HTNLearn` first encodes constraints about the methods and action models as a constraint satisfaction problem, and then solves the problem using a weighted MAX-SAT solver. `HTNLearn` can learn methods and action models simultaneously from partially observed plan traces (i.e., plan traces where the intermediate states are partially observable). We test `HTNLearn` in several HTN domains. The experimental results show that our algorithm `HTNLearn` is both effective and efficient.

Keywords: action model learning, HTN planning, learning HTNs, Weighted MAX-SAT

1. Introduction

Hierarchical Task Network (HTN) planning is an effective yet knowledge intensive method for problem-solving [66, 19]. HTN planning has been successfully used in a number of real-world applications [19, 61, 62, 13]. It requires as input both *methods*

*Corresponding Author
Email addresses: `zhuohank@mail.sysu.edu.cn` (Hankz Hankui Zhuo), `munoz@cse.lehigh.edu` (Héctor Muñoz-Avila), `qyang@cse.ust.hk` (Qiang Yang)

that describe how to decompose a task into subtasks and the preconditions under which they are applicable and *action models* that describe how actions alter the world [66, 45]. These methods and action models are traditionally encoded by human experts. This task is difficult and time-consuming even for domain experts, since it requires lots of domain knowledge-engineering effort. For example, developing the Bridge Baron[1], a commercial bridge playing algorithm that uses HTN planning techniques, took several years of development that involved major knowledge engineering effort. This suggests that it is important to develop learning algorithms to help humans build domains (i.e., action models and HTN methods) for HTN planning, aiming at saving human efforts, such as time cost, in designing HTN domains (as shown in the experiment of Section 5.3.3).

While the representation and inferencing capabilities of HTN are well understood [66, 19], the HTN knowledge has traditionally been manually defined by human experts. There has been much interest on developing learning algorithms for the task structure and applicability conditions of the HTN methods. For instance, Ilghami et al. [29] proposed an algorithm to learn preconditions of HTN methods, under the assumption that the preconditions and effects of actions in the domain were completely specified and there was complete observability of the states of the world. Xu and Muñoz-Avila [65] presented an algorithm to learn preconditions of HTN methods under the assumption that an ontology indicating relations between the objects was given. Nejati et al. [46, 51] developed approaches to learn the hierarchical structures that related the tasks and subtasks under the assumption that we have the ability to completely observe any world state and to formulate the skills to be learned in the form of Horn clauses. In [26], the HTN-MAKER algorithm learns the decomposition methods for Hierarchical Task Networks. Despite the success of current HTN learning algorithms, the above algorithms developed are all based on the assumption that the action models are known beforehand and the states are completely observable. The problem of learning the whole HTN model simultaneously, which is composed of method preconditions, method structures, and action models, has not been addressed. This problem is hard, since the information given is limited, i.e., the only knowledge provided is a set of *annotated tasks* that indicates conditions that must be held before and after the task is executed and a collection of plan traces, some of which might be annotated with nonempty *partial decomposition trees* [2]. A *decomposition tree* denotes the structure of decomposing a task into subtasks, each which is in turn further decomposed until primitive tasks are obtained. All tasks and conditions in a decomposition tree are fully grounded. A *partial* decomposition tree denotes a decomposition tree with some missing subtasks and their related edges from a decomposition tree. We should not assume that such partially annotated trees are given, but we wish to take advantage of them if they are provided.[3]

[1] http://www.greatgameproducts.com/

[2] We assume that a human expert provides both (1) the "annotated tasks", that is the preconditions and the effects of the tasks, and (2) the "partial decomposition trees", that is, some of the parent-child task relationships, and some of the precedence relations between the children tasks.

[3] In our experimental evaluation, we show performance results for our learning algorithm when none/some/many/all of the plan traces are annotated with decomposition trees.

This motivates us to develop a novel HTN learning algorithm that can cope with this problem. This is also an important problem, as in many real-world applications, neither the task hierarchy nor the action model is provided. We present several examples illustrating the applicability of HTN learning.

- In software engineering, large-scale industrial projects are often composed of thousands of software artifacts, such as requirement documents, design documents, code, bug reports, test cases, etc. The goal of software traceability in software engineering is to discover and manage the relationships that exist between these software artifacts to facilitate the efficient retrieval of relevant information, which is necessary for many software engineering tasks [2, 3]. There is a correspondence between this domain and HTN planning; each artifact can be viewed as a primitive/non-primitive task, and relationships between artifacts can be viewed as HTN structures. It is not difficult to collect partial relations between tasks (i.e., partial decomposition trees) and specify annotated tasks when developing projects. As a result, our learning algorithm can help acquire the relationships between artifacts in the form of HTN task structures. Furthermore, in this domain information about the relations between pieces of software is frequently provided by the software designer.

- In the area of web service composition, many researchers have attempted to automate the process of linking hierarchical Web services to achieve complex tasks [23]. For example, consider a website providing a service of suggesting travel itineraries. This service is decomposed to smaller services, such as weather-checking service and airplane-checking service, which are provided by other websites. Likewise, airplane-checking service is decomposed to much smaller services, such as collecting airplane information and suggesting airlines. In fact web service composition with the semantic web language OWL-S can be translated into an HTN planning model [64, 32]. Standard description languages such as SOAP (Simple Object Access Protocol)[4] are used to describe hierarchical relations among services, specifically the syntax and semantics (including task relationships and service behaviors) of the services. However, although the standard protocols of SOAP can describe the syntax of the web services, it is rather difficult if we rely on web service providers to label the semantic content of each service, especially when we consider the case that the Web services may come from many different sources. However, it is feasible to check the log data and acquire a large number of examples, which can then be used to learn the task relationships and service behaviors from examples [59]. Similarly, in this scenario, we can get a large number of logs as partial decomposition trees, and use them to learn the task relationships (HTN structures) and service behaviors (or action models), even when we do not have precise descriptions of all events happening during the web service composition process.

- In the UNIX operating system, batched OS commands have hierarchical struc-

[4] http://www.w3.org/TR/soap12-part1/

tures. For example, consider command for deleting a file. This task is accomplished by executing a sequence of low-level commands, such as opening the directory of the file and backing up some content of the file. Furthermore, backup is decomposed to much smaller commands, such as create a new empty file and copy the content to the new file, and so on. If we view each batched command as a task, the relations between batched commands can be viewed as HTN structures. Building the complex hierarchical relations between commands by hand is difficult, since there are a large number of commands in OS and relations among them are very complicated. However, it is possible to collect plan traces partially annotated with state information. For example, we can collect batch commands in an operating system that indicates 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, a batch command is a list of actions. We can easily get the action schemas (action names and its parameters) by using the UNIX history command. However, we cannot get the complete intermediate state information between these commands by reading the batch command file alone [68]. We also assume that it is possible for humans to specify a set of annotated tasks. For example, for a task of deleting a file in an operating system, it is obvious that the preconditions of the task are: the file is already created and is annotated as "unneeded". The effect is that the file is deleted. Note that the task of deleting a file may require many subtasks to achieve it, e.g., *open* the directory of the file, *backup* some content of the file, etc. The preconditions and effects of a task describe only *what* it means for the task to be accomplished (e.g., what effects will be observed in the state of the world after the task is accomplished) but not *how* to accomplish the task (e.g., what subtasks or actions must be accomplished to achieve the task). With the collected plan traces and hand-written annotated tasks as inputs, it would be possible to learn the relations among tasks.

A common trait in these domains is that the intermediate states are partially observable and the action definitions are partially given. For example, in the web service composition domain, it will be difficult to have annotated all information sources and their contents or in the software engineering domain to have the formal action model definition of all actions that can be made upon the software artifacts. Hence, existing HTN learning algorithms are unsuitable for learning in these domains. We will expand this point in the related work section.

In this paper, we present a new learning algorithm, called `HTNLearn`, to learn HTN models for HTN planning. An HTN model consists of two parts: (1) a set of *HTN methods* including *method preconditions* and *method structures* (i.e., the method's task and subtasks), and (2) a set of *action models*, each of which is associated with a precondition list, an add list and a delete list. We will give a detailed description of each of these elements in the problem definition section. Learning the two parts simultaneously is a challenging task since the information given is limited, unlike some previous systems that assume (1) complete state observability of the world and (2) action models are known. It is also advantageous to learn all parts of an HTN model simultaneously since they are inter-related and thus can be optimized together. Our algorithm models the problem of learning HTN models as a maximum satisfiability

problem (or a MAX-SAT problem [7]). In our method, we first build constraints based on inputs; then we calculate their corresponding weights using a frequent set mining algorithm. After that, we solve all the weighted constraints with a weighted MAX-SAT solver [7, 36]. Finally, we generate the HTN model based on the solution to the constraint satisfaction problem.

In [69], we presented an original version of `HTNLearn`, which was called HTN-Learner. HTN-Learner can learn *method preconditions* and *action models* simultaneously, but requires that method structures be given as input; while `HTNLearn`, which extends from HTN-Learner, can learn *method structures*, *method preconditions*, and *action models* simultaneously from partially observed plan traces which are annotated with *partial* decomposition trees. In addition, we evaluate `HTNLearn` in more detail, including evaluations with respect to partiality of decomposition trees, and running times with respect to partiality of decomposition trees and human efforts saved by `HTNLearn`. The latter is an important new consideration of our work not only with respect to the original version of `HTNLearn` but also relative to other HTN learning algorithms. We envision `HTNLearn` obtaining a first draft of the HTN model, one that will be revised by the domain expert. Our experimental evaluation empirically assesses the effort for this revision.

We organize the paper as follows. We first introduce the related work in the next section, and then give the definition of our learning problem. After that, we address our main learning algorithm `HTNLearn` and evaluate `HTNLearn` in the experiment section. Finally, we conclude the paper together with the future work.

2. Related work

2.1. Action model learning

Approaches have been proposed to learn action models from plan traces automatically. The first one is to learn action models from plan traces with complete intermediate state information. Chrisman [10] demonstrated how to learn stochastic actions with no conditional effects. Gil [20] described a system called EXPO, bootstrapped by an incomplete STRIPS-like domain description with the rest being filled in through experience. Sablon and Bruynooghe [53] presented a method to learn action models from its own experience and from its observation of a domain expert. They exploit the idea of concept induction in first-order predicate logic of inductive logic programming (ILP) [42], which allows it to utilize ILP's noise-handling techniques while learning without losing representational power. Similarly, in [5], a system was presented to learn preconditions for *teleo-operators (TOP)* using ILP. The examples used require the positive or negative examples of propositions held in states just before each action's application. ILP can learn well when the positive and negative examples of states for all target actions are given. Wang [60] described an approach to automatically learn planning operators by observing expert solution traces and refine the operators through practice in a learning-by-doing paradigm. It uses the knowledge extracted when observing experts solve problems. Oates and Cohen [48] used a general classification system to learn the effects and preconditions of actions, identifying irrelevant variables. Schmill et al. [54] proposed to learn operators with approximate computation in relevant domains by assuming that the world is completely observable. Blythe et al. [6] proposed

to acquire action models by interacting with a human expert. In [49, 50], a probabilistic, relational planning rule representation was learned, which could compactly model noisy, nondeterministic action effects. Holmes and Isbell Jr. [27] modeled synthetic items based on experience to construct action models. Walsh and Littman [59] proposed an efficient algorithm for learning action schemas for describing Web services. Cresswell et al. [12] presented a system called *LOCM* designed to carry out automated induction of action models from sets of example plans. *LOCM* assumes that a planning domain consists of sets (called *sorts*) of object instances, where each object behaves in the same way as any other object in its sort [57]. Compared with previous systems, *LOCM* can learn action models with action sequences as input, and was shown to work well under the assumption that the output domain model could be represented in an object-centered representation.

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 [19, 32, 22, 17], sometimes we cannot assume that complete intermediate state information to be given. Approaches have been developed to learn action models where the intermediate state information is not completely observable such as the ARMS system [67, 68] and SLAF [1, 56, 55, 43]. ARMS (action-relation modeling system) can automatically discover action models from a set of successfully observed plan traces. Unlike previous work in action model learning, ARMS does not assume complete knowledge of intermediate states of observed plan traces. These plan traces are obtained by an agent who does not know the logical encoding of the actions and can only partially observe state information between the actions. Specifically, ARMS gathers knowledge from the statistical distribution of frequent sets of actions in plan traces. It builds a weighted propositional satisfiability problem and solves it using a weighted MAX-SAT solver. It extracts constraints from plan traces and STRIPS models by itself, and solves these constraints with a weighted MAX-SAT solver [7]. Finally, it generates STRIPS models from the output of weighted MAX-SAT. SLAF (Simultaneously Learning And Filtering) is a tractable, exact solution for the problem of identifying actions' effects in partially observable STRIPS domains. It resembles version spaces and logical filtering in that it 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. [56] proposed an efficient algorithm to learn preconditions and effects of deterministic action models. To learn complex action models, beyond STRIPS representations, Zhuo et al. [70] proposed a framework LAMP to acquire complex action models with quantifiers and implications. These works aim at learning action models rather than learning HTN models as in our work, which include, both, action models and methods.

2.2. Learning control knowledge

There has been substantial research on learning control knowledge from previous experience to speed up planning for given action models. Minton et al. [41, 40] studied explanation-based learning (EBL) and its role in improving problem solving performance through experience. Unlike inductive systems, which learn by abstracting

common properties from multiple examples, EBL systems explain why a particular example is an instance of a concept. The explanations are then converted into operational recognition rules. In order to solve problems effectively, a problem solver must be able to exploit domain-specific search control knowledge. Although previous research has demonstrated that explanation-based learning is a viable approach for acquiring such knowledge, in practice the control knowledge learned via EBL may not be useful. To be useful, the cumulative benefits of applying the knowledge must outweigh the cumulative costs of testing whether the knowledge is applicable. Estlin and Mooney [14] present the SCOPE system for learning search-control rules that improve the performance of a partial-order planner. SCOPE integrates explanation-based and inductive learning techniques to acquire control rules for a partial-order planner. Learned rules are in the form of selection heuristics that help the planner choose between competing plan refinements. Simpson et al. [52, 39] describe a Graphical Interface for Planning with Objects called (GIPO) that have been built to investigate and support the knowledge engineering process in the building of applied AI Planning systems. Winner and Veloso [63] present a DISTILL system to learn program-like plan templates from example plans. These templates can be used to generate new plans. The focus of [63] is on how to extract plan templates from example plans as a substitute for planners. The plan templates represent the structural relations among actions. Lau et al. [33] proposed to solve a programming by demonstration (PBD) problem using a version-space learning algorithm by acquiring the normal behavior in terms of repetitive tasks. When a user is found to start a repetitive task of going through a sequence of states, the system can use the learned action sequence to map from initial to goal states directly. Marthi et al. [38] proposed semantics for high-level actions that supported proofs that a high-level plan will (or will not) achieve a goal, without first reducing the plan to primitive action sequences. Baum et al. [4] present a synthesis of two approaches, including a structure-based approximation of uniform abstraction and a dynamic locality-based approximation of envelope methods. Their solution is based on selectively ignoring some of the dimensions in some parts of the state space in order to obtain an approximate solutions with a lower computational cost. The idea of learning high-level plans in these approaches is related to learning HTNs. However, these works focus on learning control policies to speed-up the search process when solving planning problems; in principle, these problems could be solved without the learned control policies - albeit inefficiently. In contrast, in our work we are acquiring planning knowledge for solving planning problems; without this learned knowledge these planning problems cannot be solved. Another difference is that in our work we are learning planning knowledge from partial state information and incomplete traces.

2.3. Learning macro-operators

Another type of control knowledge is learning macro-operators. Macro operators are sequences of actions that can be reused as if they are a single planning operator. In planning, the use of macro-operators can significantly reduce the planning effort. Fikes et al. [16] propose to build generalized plans and use them as macro-operators and monitor plan execution. Korf [30] proposes to learn efficient strategies for solving difficult problems by searching macro-operators. Iba [28] develops a system, called MACLEARN, to learn new macro-operators, which can be defined based on other

macro-operators, to improve the efficiency of problem solving. MARVIN [11] generates macros from the plan of a reduced version of the given problem after eliminating symmetries, which learns macros from action sequences that lead its FF style search to successfully escape plateaus. Botea et al. [8] present and compare two automated methods that learn relevant information from previous experience in a domain and use it to solve new problem instances by lifting partial-order macros from plans based on an analysis of causal links between successive actions. Without focusing on exploiting particular planner or domain properties, Newton et al. [47] propose an offline method that learns macros genetically from plans for arbitrarily chosen planners and domains. He et al. [21] presented a POMDP algorithm for planning under uncertainty with macro-actions that automatically constructed and evaluated open-loop macro-actions within forward-search planning, where the planner branches on observations only at the end of macro-actions. Macro-operators can be seen as primitive HTNs. That is, HTNs decomposing non-primitive tasks into primitive tasks. We are interested in learning general HTNs including situations in which non-primitive tasks are decomposed into non-primitive tasks. This is an important distinction as illustrated by the fact that when restricted to STRIPS planning, macro-operators (e.g., as used in [30]) are strictly equivalent to STRIPS planning whereas HTN planning is strictly more expressive than STRIPS planning [**?**]. That is, there are problems that can be expressed as HTN planning problems that cannot be expressed as STRIPS planning problems. One such problem is to find the intersection of two context-free grammars.

2.4. HTN learning

Another related work is learning HTN conditions or decomposition structures. Garland et al. [18] presented an approach implemented in a development environment for constructing and maintaining a hierarchical task model from a set of annotated examples provided by domain experts, where the task model constructed did not include preconditions or effects, i.e., without methods' preconditions, actions' preconditions or actions' effects. Ilghami et al. [29] and Xu and Muñoz-Avila [65] proposed eager (in the form of version spaces) and lazy learning (in the form of case-based reasoning) algorithms respectively to learn the preconditions of HTN methods, given as input the hierarchical relationships between tasks, the action models, and a complete description of the intermediate states. Nejati et al. [46, 51] used means-end analysis to learn structures and preconditions of the input plans, assuming that a model of the tasks in the form of Horn clauses was given. Hogg et al. [26] presented an algorithm, called HTN-MAKER, to learn structures by assuming that annotated tasks are given in the form of preconditions and effects (we made the same assumption in our work). HTN-Learner, which was presented in our previous work [69], can learn *method preconditions* and *action models* simultaneously, but requires that method structures be given as input. Despite the success of previous systems, none of them can simultaneously learn method preconditions, method structures and action models.

There are also some algorithms designed to learn HTNs in nondeterministic planning domains where actions might have multiple outcomes [24], or learn probabilistic HTNs that capture user preferences by examining user-produced plans [37]. Hogg et al. [25] presented an algorithm that integrated HTN learning with Reinforcement

Learning to learn HTN methods for planning. These approaches assume that the action model is given.

To give an intuitive picture of the difference between our algorithm `HTNLearn` and previous learning systems, we list the systems as well as what can be learned by these systems in Table 1 (the relations among these systems can also be found in Figure 1). Note that we omit systems that aim to learn HTNs in nondeterministic domains in Table 1. The first row of Table 1 is our algorithm `HTNLearn` presented in this article. We can see that `HTNLearn`, which is designed based on HTN-Learner [69], is able to learn method structures, method preconditions and action models simultaneously, while other systems only learn parts of them.

Table 1: The difference of the current systems.

algorithms	what can be learned
`HTNLearn` [this paper]:	(I)(II)(III)
HTN-Learner[69]:	(II)(III)
HTN-MAKER[26]:	(I)(II)
Garland et al. [18]:	(I)
CaMeL[29], DInCAD[65], Icarus[46], XLearn[51]:	(II)
ARMS[68], SLAF [1]:	(III)

(I): *method structures*, (II): *method preconditions*, (III): *action models*.

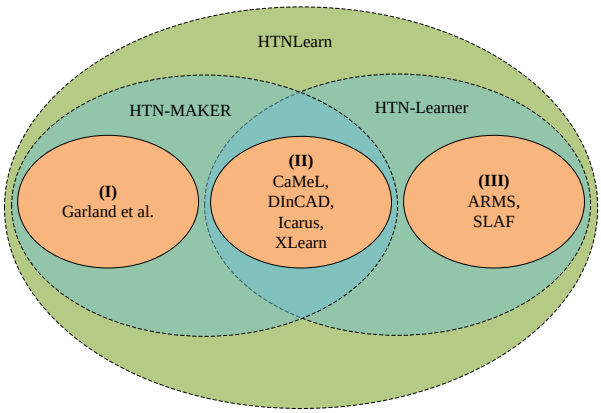


Figure 1: Relations among different systems with respect to what can be learned.

3. Problem definition

We are focusing on a variant of HTN planning called Ordered Task Decomposition ([44] and Chapter 11 of [19]), which is the most commonly used variant of HTN planning. In this variant the planning system generates a plan by decomposing tasks in the order they will be later executed. Tasks are decomposed into simpler and simpler subtasks until primitive tasks are reached, which 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 [15]. 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. There have been some planners proposed to solve HTN planning problems, e.g., Nau et al. [45] developed a domain-independent system, called SHOP, to solve HTN planning problems, and Kuter et al. [31] presented a planning algorithm, called Yoyo, for solving planning problems in fully observable nondeterministic domains by combining an HTN-based mechanism for constraining its search and a Binary Decision Diagram (BDD) [9] representation for reasoning about sets of states and state transitions.

A Hierarchical Task Network (HTN) planning problem can be defined as a quadruple (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 *HTN 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 correspond to primitive subtasks that can be directly executed [66, 19].

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. An *annotated task* [26] is defined by $(t, \text{PRE}, \text{EFF})$, where PRE and EFF are sets of preconditions and effects of task t, respectively. The preconditions and effects of an annotated task indicate conditions that must be satisfied in the states immediately before and immediately after achieving the task. 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 m decomposes, PRE is a list of preconditions of the method, and SUB is a list of subtasks into which the head task t may be decomposed. The arguments of m consist of the arguments of the head task and the subtasks. Subtasks may be primitive or non-primitive. A primitive task correspond to an *action schema*. An action schema is composed of an action name and zero or more arguments. A non-primitive task must be further decomposed. Each precondition of an HTN method is a literal, and the set of preconditions must be satisfied before the method can be applied to decompose a non-primitive task. We define a *method precondition* as a triple (m, t, PRE), and a *method structure* as (m, t, SUB). An *action model* a is composed of a quadruplet $(o, \text{PRE}, \text{ADD}, \text{DEL})$, where o is an action schema, PRE is a *precondition list*, ADD is an *add list* and DEL is a *delete list*, which follows the STRIPS description [15].

A *solution* to an HTN planning problem (s_0, T, M, A) is a list of *decomposition trees* [66, 19]. There is one tree for each task t in T and the root of each tree is the corresponding task t. Decomposition trees indicate how each task in T is achieved. The nodes of a decomposition tree are tasks. Interior nodes contain non-primitive tasks. Their children are the subtasks of the applicable method used to decompose the non-primitive task. Leaf nodes contain primitive tasks. A leaf node is a completely instantiated action; the list of actions $\langle a_1, a_2, \ldots, a_n \rangle$ in the tree can be directly executed from the initial state to accomplish the root level task. We denote by $\text{EXEC}(i)$ the state between a_i and a_{i+1}, which represents the state after executing actions $\langle a_1, a_2, \ldots, a_i \rangle$.

An example of decomposition tree is shown in Figure 2. This tree includes two non-

primitive tasks: *'(remove ?x)'* and *'(superpose ?x ?y)'*. The task ***'(remove ?x)'*** moves to the table the whole stack whose bottom block is '?x', i.e., the order of blocks above '?x' (including '?x') remains unchanged; the task ***'(superpose ?x ?y)'*** stacks block '?x' on top of the block '?y', i.e., the order of blocks above '?x' (including '?x') remains unchanged; 'unstack', 'putdown', 'pickup' and 'stack' are four actions to un-stack, put down, pickup and stack a block. Figure 2(a) depicts the initial state. Figure 2(b) depicts the goal state. By decomposing the task "(remove B)" into subtasks (and further decomposing these subtasks into smaller subtasks) using decomposition methods, as is shown in Figure 2(c), the planner can achieve the task.

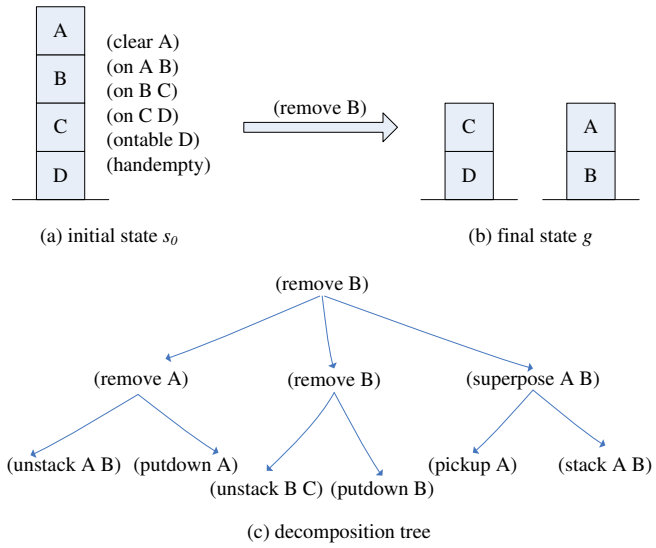


Figure 2: An example decomposition tree

When some tasks (i.e., the root or interior nodes) and their *related* edges are missing in a decomposition tree, we refer to it as a *partial* decomposition tree. A set of partial decomposition trees is denoted by Π^{part}. Tasks' related edges include edges between themselves and their parents, themselves and their children, and their parents and their siblings. An example of a partial decomposition tree is shown in Figure 3. The dashed lines indicate missing parts of the tree. Partial decomposition trees can be empty, which corresponds to the scenario that all the interior nodes and root are missing. For brevity, we will also use the term "partial decomposition trees" to refer to complete decomposition trees, which indicate no tasks or edges are missing.

The HTN learning problem. Given as input: (1) a set of partial decomposition trees Π^{part} annotated with partially observed states, and (2) a set of annotated tasks $\mathcal{T}$ for an unspecified HTN planning domain, obtain: (1) a set of HTN methods (including *method preconditions* and *method structures*) and (2) a set of action models that is consistent the inputs.

An example of the input is shown in Figure 4, where there are partially observed intermediate states s_i between leaves of partial decomposition trees. A partially ob-

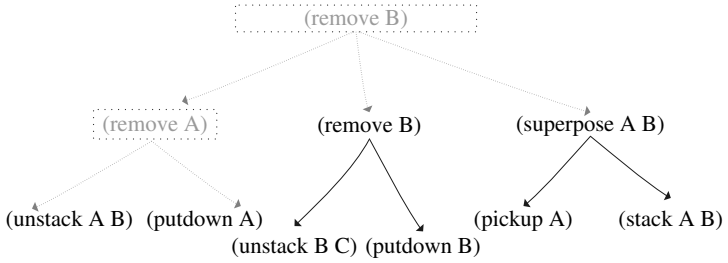
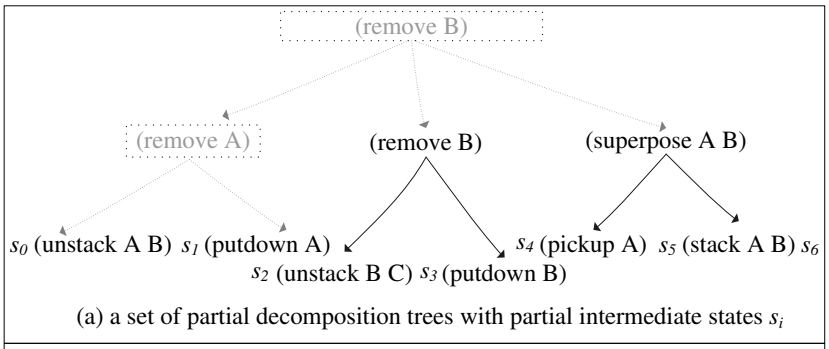


Figure 3: An example partial decomposition tree

served state indicates that some of the propositions are missing in a state. For example, $s_1 = \{$*(holding A)(clear B)*(on B C) (on C D) (ontable D)$\}$ is a partially observed state, where the gray parts indicates they are missing. An example output of our algorithm is shown in Table 2, where the task "(remove ?x)" is decomposed into the subtasks "(unstack ?x ?y)" and "(putdown ?x)" by the method called "(remove-m0 ?x ?y)", and into subtasks "(remove ?x)", "(remove ?y)" and "(superpose ?x ?y)" by the method called "(remove-m1 ?x ?y)". We name a method that is applied to a task called "⟨*task name*⟩" as "⟨*task name*⟩-mX", where "X" indicates the index of methods that can be applied to the same tasks with different preconditions. The parameters of a method are composed of all the parameters of the task and its subtasks.



(a) a set of partial decomposition trees with partial intermediate states s_i

(:task (remove ?x)
 :**preconditions** (and (handempty))
 :**effects** (and (ontable ?x)))
(:task (superpose ?x ?y)
 :**preconditions** (and (handempty) (ontable ?x) (clear ?y))
 :**effects** (and (on ?x ?y)))
… (other annotated task omitted)

(b) a set of annotated tasks

Figure 4: An example input of `HTNLearn` (s_0 is the initial state: $s_0 = \{$(on A B) (on B C) (on C D) (ontable D) (clear A) (handempty)$\}$; s_1, s_2, s_3, s_4, s_5 and s_6 are partially observed states.)

Table 2: An example output of HTNLearn

methods	(:method (remove-m0 ?x - block ?y - block) :**task** (remove ?x) :**preconditions** (and (handempty) (on ?x ?y) (clear ?x)) :**subtasks** (and (unstack ?x ?y) (putdown ?x))) (:method (remove-m1 ?x - block ?y - block) :**task** (remove ?y) :**preconditions** (and (handempty) (on ?x ?y)) :**subtasks** (and (remove ?x) (remove ?y) (superpose ?x ?y))) ...
actions	(:action (unstack ?x - block ?y - block) :**preconditions** (and (handempty) (on ?x ?y) (clear ?x)) :**effects** (and (holding ?x) (clear ?y) (not (handempty)) (not (on ?x ?y)) (not (clear ?x))) (:action (putdown ?x - block) :**preconditions** (holding ?x) :**effects** (and (not (holding ?x)) (clear ?x) (ontable ?x) (handempty))) (:action (pickup ?x - block) :**preconditions** (and (handempty) (clear ?x) (ontable ?x)) :**effects** (and (holding ?x)(not (handempty)) (not (clear ?x)) (not (ontable ?x)))) (:action (stack ?x - block ?y - block) :**preconditions** (and (holding ?x) (clear ?y)) :**effects** (and (handempty) (clear ?x) (on ?x ?y) (not (holding ?x)) (not (clear ?y))))

Our learning method is designed to solve the above-mentioned learning problem for which annotated tasks are given by the user. We can learn decomposition methods for HTNs when their intended tasks are annotated. Since the annotated tasks are defined as preconditions and effects, the methods we can learn are restricted to those models expressible using STRIPS representations. However, our definition does not preclude that more complex HTN models are elicited. For example, a user might provide an incomplete set of tasks and methods that contain intrinsic problem structure that is not expressible in STRIPS representations and the user provides annotated tasks to fill the gaps in the HTN model. Hence, while those gaps (or annotated tasks) will need to be equivalent to STRIPS representations, the overall elicited model can be more complex, i.e., including intrinsic structures and preconditions/effects representations.

4. The HTNLearn algorithm

In this section, we present the algorithm for learning HTN from partial decomposition trees. In particular, we first present an overview of the algorithm HTNLearn in subsection 4.1, and then provide a detailed description of each step in subsections 4.2-4.8.

4.1. Algorithm framework

HTNLearn takes as input a set of annotated tasks and a set of partial decomposition trees annotated with partially observed intermediate states. It outputs a collection of action models and a collection of methods. A key point of HTNLearn is its ability to learn with partial information; namely, HTNLearn copes with situations where only partially observed states and partial decomposition trees are given. To achieve this, HTNLearn constructs constraints (weighted by the frequency of their occurrences) that reflect the information as it becomes available. Specifically, it builds the following kinds of constraints from the given input:

- **Method constraints** encode the information about the method preconditions and method structures.
- **State constraints** encode information about the action models.
- **Decomposition constraints** and **task constraints** enhance the learned HTN models using statistical information from the partial decomposition trees
- **Hard constraints** ensure the learned HTN models are consistent with the inputs.

Based on these constraints, HTNLearn builds a set of clauses to be solved as a weighted maximum satisfiability problem. We use a weighted MAX-SAT solver for this purpose [35]. The solution to this MAX-SAT problem is the HTN model including the set of action models and HTN methods that best explains the given inputs. An overview of the algorithm is shown in Algorithm 1.

Algorithm 1: Algorithm overview of HTNLearn

Input: A set of partial decomposition trees Π^{part} annotated with partially observed states and a set of annotated tasks $\mathcal{T}$.
Output: The HTN model $\mathcal{H}$;

1: Extract predicates P and action schemas A;
2: Build a set of method constraints MC;
3: Build a set of state constraints SC;
4: Build a set of decomposition constraints DC;
5: Build a set of task constraints TC;
6: Build a set of hard constraints HC, including partialness constraints, action constraints and plan constraints;
7: Solve all the constraints using a weighted MAX-SAT solver [36], and convert the solution to the HTN model $\mathcal{H}$;
8: **return** $\mathcal{H}$;

4.2. Step 1: extract predicates and action schemas

Action models and methods are defined in terms of predicates. In addition, action models are described by action schemas. In this step, we extract predicates and action schemas from the given partial decomposition trees annotated with the observed intermediate states. We do this extraction in the following two steps.

1. We scan all the states including initial, intermediate and goal states, and select the predicates corresponding to the variable versions of propositions (i.e., substituting instantiated parameters (objects) of propositions with variables) in each state. For example, by scanning the initial state s_0 ={(on A B) (on B C) (on C D) (ontable D) (clear A) (handempty)} in Figure 4, we attain the following predicates (we delete all the duplicate predicates): {(on ?x - block ?y - block) (ontable ?x - block) (clear ?x - block) (handempty)}. We assume that the *type* of each object is known. When converting an object to a variable, we associate the variable with the object's type,[5] e.g., since the type of object "A" is known as "block", we associate its corresponding variable "?x" with "block".
2. We scan all the instantiated actions from the leaves of the partial decomposition trees, and attain action schemas by substituting all the instantiated parameters with variables. For example, by scanning the leaves of the partial decomposition tree in Figure 4, we can extract the following action schemas: {(unstack ?x - block ?y - block) (putdown ?x - block) (pickup ?x - block) (stack ?x - block ?y - block)}.

4.3. Step 2: build method constraints

In this step, we encode the structure information from the given partial decomposition trees and annotated tasks. Intuitively, if the preconditions of an annotated task t are included in a state s_i (before the execution of action a_i), and the effects of t are included by a state s_j (after the execution of action a_j), t is likely achieved by the action sequence $\langle a_i, \ldots, a_j \rangle$, where $i \leq j$. We call the structure $\langle t, a_{[i,\ldots,j]} \rangle$ a candidate *primitive hierarchy* (see Definition 1). The set of which is denoted by $\mathcal{H}^{cand}$. Furthermore, if another annotated task t' is achieved by a subsequence of $\langle a_i, \ldots, a_j \rangle$, we can conjecture that t' is one of the descendants of t, i.e., t' is either a subtask of t, or, recursively, a subtask of a subtask descendant of t. In this way, we can build a set of candidate *non-primitive hierarchies* (see Definition 2), denoted by $\mathcal{H}^{non}$. Furthermore, to ensure the learned method structures are succinct, we filter unneeded non-primitive hierarchies (i.e., hierarchies that can be inferred by other hierarchies in $\mathcal{H}^{non}$) from $\mathcal{H}^{non}$, resulting in a set of candidate method structures, denoted by $\mathcal{S}^{cand}$. Based on $\mathcal{S}^{cand}$, we build a set of *method constraints* to capture the final set of method structures. In the following, we first give the definitions of *primitive hierarchy* and *non-primitive hierarchy*, and then describe how to generate candidate primitive hierarchies and candidate method structures, and finally present method constraints.

Definition 1. [**Primitive Hierarchy**] *A primitive hierarchy is a tuple $\langle t, a_{[1,\ldots,k]} \rangle$, where t is a task, and $a_{[1,\ldots,k]}$ is an action sequence $a_1, \ldots, a_k$ generated by decomposing task t with one or more decomposition methods. Note that t can be a primitive task, in which case k is 1.*

As an example, in Figure 2, "⟨(remove B), [(unstack A B) (putdown A) (unstack B C) (putdown B) (pickup A) (stack A B)]⟩" is a primitive hierarchy.

[5] When there is a type hierarchy, we associate variables with the lowest level of the hierarchy that is consistent with all relevant objects in the plan traces.

Definition 2. [**Non-Primitive Hierarchy**] *A tuple $\langle h, h_{[1,\ldots,m]}\rangle$, where h and h_i are primitive hierarchies, is called a non-primitive hierarchy, if the action sequence of h is the same as the result of concatenating all the action sequences of $h_1, \ldots, h_m$ in order.*

In order to encode the hierarchical relationship among tasks, we build a set of candidate primitive hierarchies $\mathcal{H}^{cand}$, which serves as a set of basic structures for building method constraints. We do this by simply scanning each partial decomposition tree in Π^{part} and each annotated task in $\mathcal{T}$ and exploiting their relations between states and task preconditions and task effects to build $\mathcal{H}^{cand}$. The detailed description is given in Algorithm 2.

Algorithm 2: generate candidate primitive hierarchies: *gen-prm-hier*(Π^{part},$\mathcal{T}$)

Input: a set of partial decomposition trees Π^{part}, a set of annotated tasks $\mathcal{T}$;
Output: a set of candidate primitive hierarchies $\mathcal{H}^{cand}$;

1: $\mathcal{H}^{cand} = \emptyset$;
2: **for** each partial decomposition tree $\pi \in \Pi^{part}$ **do**
3: *denote the leaves of π with partial states as* $(s_0, a_1, s_1, \ldots, a_n, s_n)$;
4: **for** each annotated task in $t \in \mathcal{T}$ **do**
5: **for** $i = 0$ **to** $n - 1$ **do**
6: **if** $conflict(s_i, \text{PRE}(t))$=false **then**
7: **for** $j = i + 1$ **to** n **do**
8: **if** $conflict(s_j, \text{EFF}(t))$=false **then**
9: *generate a new candidate primitive hierarchy* $h = \langle t, a_{[i+1,\ldots,j]}\rangle$;
10: *add h to $\mathcal{H}^{cand}$:* $\mathcal{H}^{cand} \leftarrow h$;
11: **end if**
12: **end for**
13: **end if**
14: **end for**
15: **end for**
16: **end for**
17: **return** $\mathcal{H}^{cand}$;

In step 6 of Algorithm 2, the procedure $conflict(s_i, \text{PRE}(t))$ returns *true* if there is a proposition in s_i contradicting some literal in PRE(t). For example, if there exists $p \in P$ such that $\neg p \in s_i$ and $p \in \text{PRE}(t)$, then they are contradicting one other, and the procedure returns *true*. Otherwise, it returns *false*.

Furthermore, using the candidate primitive hierarchies $\mathcal{H}^{cand}$ generated by Algorithm 2, we generate a set of candidate method structures $\mathcal{S}^{cand}$. The procedure of generating candidate method structures can be found in Algorithm 3. In step 2 of Algorithm 3, we build all possible non-primitive hierarchies by simply scanning different combinations of primitive hierarchies in $\mathcal{H}^{cand}$. In steps 3-7, we choose those non-primitive hierarchies that have no subsequences of primitive hierarchies occurring in other non-primitive hierarchies, and store them in $\mathcal{S}^{cand}$. Specifically, given three non-primitive hierarchies $g_1 = \langle h_a, h_{[1,\ldots,m]}\rangle$, $g_2 = \langle h_b, h_{[x,\ldots,y]}\rangle$ and $g_3 = \langle h_a, h_{[1,\ldots,x-1,b,y+1,\ldots,m]}\rangle$, where h_a and h_b are two primitive hierarchies differ-

ent from $h_1, \ldots, h_m$, and $1 \leq x \leq y \leq m$, we will delete g_1 and just keep g_2 and g_3 in $\mathcal{S}^{cand}$ because g_1 can be generated by applying g_2 and g_3.

Algorithm 3: generate candidate method structures: *gen-mth-str*($\mathcal{H}^{cand}$)

Input: A set of candidate primitive hierarchies $\mathcal{H}^{cand}$;
Output: A set of candidate method structures $\mathcal{S}^{cand}$;

1: $\mathcal{S}^{cand} = \emptyset$;
2: *generate a set of all possible non-primitive hierarchies using* $\mathcal{H}^{cand}$, denoted by $\mathcal{H}^{non}$;
3: **for** each non-primitive hierarchy $g = \langle h, h_{[1,\ldots,m]} \rangle \in \mathcal{H}^{non}$ **do**
4: **if** there is no other non-primitive hierarchy $g' = \langle h', h_{[x,\ldots,y]} \rangle \in \mathcal{H}^{non}$, where $1 \leq x \leq y \leq m$ **then**
5: *add* g *to* $\mathcal{S}^{cand}$: $\mathcal{S}^{cand} \leftarrow g$;
6: **end if**
7: **end for**
8: **return** $\mathcal{S}^{cand}$;

It is possible that Algorithm 3 will generate an exponential number of methods on the number of actions in the partial decomposition tree (i.e., the input for Algorithm 2). Such situations can happen depending on the definitions of the given annotated tasks; annotated tasks may make multiple groupings of the actions possible, which in turn may result in multiple possible groupings of the parent tasks at the next level, and subsequently propagate this combinatorial effect to higher levels of the hierarchy. As we will see in the experiments, this is not the case for the particular domains and the particular annotated tasks used. But if such a situation were to occur, Algorithms 2 and 3 can be easily modified to reduce the number of methods generated by restricting the types of groupings allowed (e.g., always grouping the largest possible sequences of actions/tasks).

Using $\mathcal{H}^{cand}$ and $\mathcal{S}^{cand}$, we build constraints as follows. We denote the final set of primitive hierarchies as $\mathcal{H}$, and the final set of method structures as STR.

1. For each $h = \langle t, a_{[i,\ldots,j]} \rangle \in \mathcal{H}^{cand}$, if the following constraints are satisfied,

$$\text{PRE}(t) \subseteq \text{EXEC}(i-1) \wedge \text{EFF}(t) \subseteq \text{EXEC}(j)$$

   we conjecture that $h \in \mathcal{H}$. Note that $\text{EXEC}(i-1)$ represents the state after executing actions $a_1, \ldots, a_{i-1}$.
2. For each $\langle h, h_{[1,\ldots,m]} \rangle \in \mathcal{S}^{cand}$, if the following constraints are satisfied,

$$h \in \mathcal{H} \wedge \bigwedge_{1 \leq i \leq m} h_i \in \mathcal{H}$$

   we conjecture that $\langle t, t_{[1,\ldots,m]} \rangle \in \text{STR}$, where t is the task being decomposed in h, and t_i is the task being decomposed in h_i ($1 \leq i \leq m$).

We build the constraints for creating the method structures STR iteratively based on the above two cases. To generate a method m_k for a method structure in STR, we

set m_k's parameters to be all the different parameters of the tasks in the method structure. We also need to build constraints for generating m_k's preconditions. We exploit a straightforward way to generate the preconditions by assuming that, if a predicate $p \in P$ frequently appears in the state where the method m_k is executed and its parameters are included in m_k, then p is one of m_k's preconditions. We denote the method structure's corresponding primitive hierarchy by $\langle t, a_{[i,\ldots,j]} \rangle$. We capture this idea with the following constraint (note that the constraint is defined for all predicates $p \in P$ and all methods m_k),

$$(p \in \text{EXEC}(i-1) \wedge (\text{PARA}(p) \subseteq \text{PARA}(m_k))) \\ \Rightarrow p \in \text{PRE}(m_k)$$

where PARA(p) and PARA(m_k) are the sets of p and m_k's parameters respectively. Note that $p \in$ EXEC$(i-1)$ means p either exists in the initial state and is not deleted by the action sequence $\langle a_1, \ldots, a_{i-1} \rangle$, or is added by some action a' prior to a_i and is not deleted by actions between a' and a_i. With the precondition list PRE(m_k) and the structure STR$(m_k) \in$ STR, we can build a set of method constraints MC as follows:

$$\langle m_k, \text{PRE}(m_k), \text{STR}(m_k) \rangle \in \mathcal{M},$$

where $\mathcal{M}$ denotes a set of methods.

We have thus built the set of all the method constraints MC for creating a set of methods $\mathcal{M}$ based on the given plan traces. We use the procedure "*calculate_weights*(MC)" to combine all the instantiated constraints into their corresponding variable-form constraints and calculate their weights. The intuition behind this procedure is that if a constraint is more frequently satisfied in the plan traces, this constraint is more likely to be true. This is the same intuition that MAX-SAT uses for the weights, i.e., the larger the weight is, the more likely the constraint will be satisfied (a similar idea is used in [68]). This procedure takes place in the following two steps.

1. We replace all the instantiated arguments in MC with their corresponding variables, and denote the result as MC$'$.
2. We calculate the weighted method constraints (denoted as WMC) by

$$\text{WMC} = \{\langle w, f \rangle | f \in \text{MC}' \wedge w = countNum(f, \text{MC}')\},$$

where $countNum(f, \text{MC}')$ returns the number of occurrences of f in MC$'$.

We assume that two methods m_1 and m_2 with the same STR (i.e., STR$(m_1) =$ STR(m_2)) can be combined into a single method, denoted by m', whose preconditions are composed of the common preconditions between m_1 and m_2 (i.e., PRE$(m') =$ PRE$(m_1) \cap$ PRE(m_2)). This is consistent with the definition of the methods' preconditions (i.e., preconditions are conjunctive). Since m_1 and m_2 can be viewed as a method with two disjunctive preconditions PRE(m_1) and PRE(m_2), in all cases that m_1 and m_2 are applicable, their combined method m' is also applicable. This suggests it is reasonable to use m' instead of m_1 and m_2 since m' cover all the cases that either m_1 or m_2 cover in the given decomposition trees.

4.4. Step 3: build state constraints

The sets of conditions, i.e, actions' preconditions, actions' effects and methods' preconditions, are generated based on the following observations: (1) if a predicate frequently appears before an action a is executed, and its parameters are also the parameters of a, then the predicate is likely to be a precondition of a. Likewise, (2) if a predicate frequently appears before a method m is applied, it is likely to be a precondition of m; (3) if a predicate frequently appears after a is executed, it is likely to be an effect of the a. This information is 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 the set SC of state constraints (PARA(p) denotes the set of parameters of p):

1. For each predicate $p \in P$ in the state where an action a is executed and PARA(p) $\subseteq$ PARA(a), we build a constraint $p \in$ PRE(a). This constraint indicates that predicate p might be a precondition of action a. We denote the set of these constraints as SC$_{pa}$.
2. For each predicate p in the state after an action a is executed such that PARA(p) $\subseteq$ PARA(a), we build a constraint $p \in$ ADD(a). This constraint indicates that predicate p might be an effect of action a. We denote the set of these constraints as SC$_{ap}$.
3. For each predicate p in the state where a method m is applied, we build a constraint $p \in$ PRE(m). This constraint indicates that predicate p might be a precondition of method m. We denote the set of these constraints as SC$_{pm}$

As a result of Step 3, we generate three kinds of constraints SC$_{pa}$, SC$_{ap}$ and SC$_{pm}$, which are put together to form the state constraints SC. With SC, we build weighted state constraints WSC with the procedure "*calculate_weights*(SC)", which is presented in the end of Section 4.3 (i.e., using MC instead of SC).

4.5. Step 4: build decomposition constraints

In this step, we build decomposition constraints to encode the structure information provided by decomposition trees. (1) If a task T can be decomposed into n subtasks st_1, st_2, ..., st_n, we find that the sub-tree of the decomposition tree whose root is subtask st_i often provides some of the preconditions of the method m_{t+1} decomposing subtask st_{i+1}. In addition, (2) we find that the parameters of a precondition should be included in the list of parameters of the action or method the precondition belongs to. Analogously, (3) the parameters of an effect should be included in the list of parameters of the action that the precondition belongs to. The decomposition constraints DC are built based on these three assumptions by the procedure shown in Algorithm 4.

In the fourth step of Algorithm 4, we consider two tasks st_i and st_j that have the same parent and such that 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 $= \{p|$PARA(p) $\subseteq$ PRS $\wedge\, p \in P\}$. In the last step, the generated constraint c is:

$$p \in \text{GP} \rightarrow (p \in \text{ADD}(a_{ik}) \wedge p \in \text{PRE}(m_j)).$$

Algorithm 4: Build decomposition constraints: *buildDecmpConstr(*Π^{part}*)*

Input: A set of decomposition trees with partially observed states Π^{part};
Output: Decomposition constraints DC;

```
DC = ∅;
for each decomposition tree dtr ∈ Π^part do
    for each two subtasks st_i and st_j in dtr do
        if i < j and there is a method m_j in dtr decomposing st_j then
            for k = 1 to n_i (n_i is #actions to accomplish the subtask st_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;
```

With DC, we build the set of weighted decomposition constraints WDC using the procedure "*calculate_weights*(DC)", which is presented in the end of Section 4.3 (using MC instead of DC).

4.6. Step 5: build task constraints

If a task is directly decomposed into a list of actions by a certain HTN method, then these actions may have strong relations with each other. Specifically, an action a often provides a predicate for a's succeeding action b, such that b can be executed (the predicate is one of b's preconditions). For all k ($i \leq k < j$), we formulate these task constraints as follows:

$$\langle t, a_{[i,\ldots,j]} \rangle \in \text{STR} \wedge p \in \text{ADD}(a_k) \wedge (\text{PARA}(p) \subseteq \text{PARA}(a_{k+1})) \Rightarrow p \in \text{PRE}(a_{k+1}).$$

We call this kind of constraints *task constraints* and denote them as TC. We calculate the weighted task constraints WTC with the procedure "*calculate_weights*(TC)", which is presented in the end of Section 4.3 (using MC instead of TC).

4.7. Step 6: build hard constraints

In this step, we build three types of hard constraints (denoted by HC), i.e., partialness constraints, action constraints and plan constraints, to ensure that the learned HTN models are *consistent* with the known structures of partial decomposition trees, the STRIPS conventions and the plan traces (i.e, the leaves of the partial decomposition trees), respectively. In the following we describe each type of hard constraints in detail.

4.7.1. Partialness constraints

These constraints ensure that the learned HTN models are consistent with the given partial decomposition trees. Specifically, the method structures indicated in the partial decomposition trees should be reflected in the learned methods. For example, given the partial decomposition tree in Figure 3, the method structure $\langle$(superpose ?x ?y), [(pickup ?x), stack(?x ?y)]$\rangle$ should be learned, where ?x is a variable that is instantiated by "A", and ?y is a variable that is instantiated by "B" in Figure 3. We build a set of constraints, called partialness constraints, to ensure these method structures are learned.

We build partialness constraints by simply performing the following two steps. (1) We scan all the given partial decomposition trees and collect all method structures $g = \langle t, t_{[1,\ldots,m]} \rangle$, where t is decomposed to $t_1, \ldots, t_m$. (2) We replace all the parameters in the tasks of g with their corresponding variables, and build constraints $g \in$ STR and store them in HC.

4.7.2. Action constraints

To make sure that the learned action models are consistent with typical STRIPS conventions made when hand-writing STRIPS domains, we generate additional constraints, called action constraints. We formulate the constraints as follows [68] and store them in HC:

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 in P, ADD(a) is a set of added effects of the action a, and PRE(a) is a set of preconditions of a in 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, typically, existing planning domains follow these conventions. However, our learning algorithm can work without them.

4.7.3. Plan constraints

These constrains are in place so that the learned action models are consistent with the training plan traces. They impose a requirement on the relationship between ordered actions in plan traces, and ensure that the causal links in the plan traces are not broken. These requirements are: (1) for each precondition p of an action a_j in a plan trace, either p is in the initial state, or there is an action a_i ($i < j$) prior to a_j that adds p and there is no action a_k ($i < k < j$) between a_i and a_j that deletes p, and (2) for each literal q in a state s_j, either q is in the initial state s_0, or there is an action a_i before s_j

that adds q while no action a_k deletes q. We formulate these constraints as follows and store them in HC:

$$p \in \text{PRE}(a_j) \wedge (p \in s_0 \vee (\exists i.\forall k.((i < k < j) \wedge p \in \text{EFF}(a_i) \wedge \neg p \notin \text{EFF}(a_k))))$$

and

$$q \in s_j \wedge (q \in s_0 \vee (\exists i.\forall k.((i < k < j) \wedge q \in \text{EFF}(a_i) \wedge \neg q \notin \text{EFF}(a_k)))),$$

For example, since "holding" is a precondition of the action "putdown", either "holding" is an atom in the initial state or there is an action, such as "pickup", that adds it, and it is never deleted by any actions between "pickup" and "putdown". Otherwise, the action "putdown" cannot be executed after the action "pickup".

We require the above constraints to be *hard*, i.e., they should be satisfied to ensure that the learned HTN models are consistent with the partial decomposition trees, the STRIPS conventions and the plan traces. To do this, we assign a large enough weight, denoted by w_{max}, to these constraints. In our experiment, we simply chose the maximal weight of all the constraints in WMC, WSC, WDC and WTC as the value of w_{max}, to make the weight of hard constraints "comparable" to soft constraints (and view the soft constraints with the maximal weight as hard constraints as well).

4.8. Step 7: solve all the constraints

Following Steps 2-6, we built six types of weighted constraints to encode the information needed to generate the action models and the methods. In this step, we put all these constraints together and solve them with a weighted MAX-SAT solver.

In MAX-SAT, a proposition variable x_i may take values *false* or *true*. A literal l_i is either a variable x_i or its negation $\bar{x}_i$. A clause is a disjunction of literals, and a CNF formula ϕ is a conjunction of clauses. An assignment of truth values to the propositional variables satisfies a literal x_i if x_i takes the value *true* and satisfies a literal $\bar{x}_i$ if x_i takes the value *false*. An assignment satisfies a clause if it satisfies at least one literal of the clause, and satisfies a CNF formula if it satisfies all the clauses of the formula. An assignment for a CNF formula ϕ is complete if all the variables occurring in ϕ have been assigned; otherwise, it is partial. The MAX-SAT problem for a CNF formula ϕ is to find an assignment of values to propositional variables that minimizes the number of unsatisfied clauses (or equivalently, that maximizes the number of satisfied clauses). The MAX-SAT problem is NP-hard, since the boolean satisfiability problem, which is NP-complete, can be easily reduced into MAX-SAT. One extension to MAX-SAT is weighted MAX-SAT which asks for the maximum weight which can be satisfied by any assignment, given a set of weighted clauses. There have been many approaches that solve weighted MAX-SAT problems, such as MaxSatz [36, 35, 34], which implements a lower bound computation method that consists of incrementing the lower bound by one for every disjoint inconsistent subset that can be detected by unit propagation.

In `HTNLearn`, we exploit MaxSatz to solve all the built constraints, and attain a *true* or *false* assignment to maximally express the weighted constraints. According to the assignment, we can acquire the HTN model directly. For example, 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. Another example, if "$g \in$ STR" is assigned *true*, then g will be converted into a method structure.

5. Experiments

To evaluate the performance of HTNLearn, we design and carry out experiments in both synthetic and benchmark domains. Specifically, we evaluate the accuracy of the HTN model learned by HTNLearn,[6] estimate the human effort saved by HTNLearn and measure HTNLearn's running time. These experiments are described in the following subsections:

- Section 5.3.1 presents the accuracy results for the domains learned with HTNLearn when complete decomposition trees are provided.
- Section 5.3.2 presents the accuracy results for domains learned with HTNLearn when partial decomposition trees are provided.
- Section 5.3.3 presents results about the estimated amount of human effort that is saved when creating HTN models based on the models learned by HTNLearn.
- Section 5.3.4 presents the running time of HTNLearn when learning HTN models with respect to number of plan traces given in the input.

5.1. Datasets and experimental setup

We use the HTN domains called *htn-blocks*, *htn-depots*, *htn-driverlog* and *htn-ebusiness* for training and testing. Table 3 presents the number of actions, tasks and methods in these domains.

Table 3: HTN domains used in the experiments

domains	actions	tasks	methods
htn-blocks	4	3	10
htn-driverlog	6	4	10
htn-depots	6	8	13
htn-ebusiness	4	3	5

htn-blocks: In Table 3, *htn-blocks* is created based on the *blocks world* domain from IPC-2 ,[7] which is composed of four actions (primitive tasks) as is given in the *blocks world* domain; and three tasks describing procedures that reverse two blocks in the same or different stacks, remove a block in a stack, and superpose a block on another block, respectively; and ten methods that decompose tasks into subtasks.

[6]The HTNLearn system, as well as testing data, can be downloaded from the website: http://www.zsusoft.com/%7ehankz/downloads/downloads.htm.

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

htn-driverlog: The domain *htn-driverlog* is created based on the domain *driverlog* from IPC-3,[8] which is composed of six actions, four tasks and ten methods. The six actions have the same meanings as described in the IPC-3's *driverlog* domain. The four tasks covers procedures that move an object from one location to another, schedule a truck to a target location, drive a truck from one location to another, and tell a driver to walk to a target location, respectively. The ten methods are used to decompose tasks to subtasks including primitive tasks.

htn-depots: The *htn-depots* domain is also created based on the domain *depots* from IPC-3, which is composed of: six actions given by IPC-3's *depots*; eight tasks to move a crate from one place to another, clean crates above some crate, remove a crate as well as crates above the crate, superpose a crate (as well as crates above the crate) on another crate, install a crate into a truck, uninstall a crate from a truck, get a hoist to a target place, and drive a truck to a target domain. There are also thirteen methods to decompose tasks.

htn-ebusiness: In the last row of Table 3, *htn-ebusiness* is created based on the simplified processes of electronic business. It consists of four actions specifying buying goods from a sales network, scheduling goods from a warehouse to another, transporting goods from a warehouse to a client, and paying for goods, respectively. There are three tasks covering the procedures that a client shops goods from sales network (must get the goods successfully), goods are transported from one warehouse to another, and goods are transported from a warehouse to a client, respectively. There are also five methods that decompose tasks to subtasks.

For each of these domains, we use a version of the domain consisting of hand-crafted methods and operators. This version is our *ground-truth* model. We generate 200 plan traces and their decomposition trees using this version of the domain. We gave these 200 plan traces, a subset of the decomposition trees (see below for details), and a collection of annotated tasks as input to `HTNLearn`. We then compare the resulting domain with the ground-truth model.

5.2. Evaluation metrics

For the `HTNLearn` algorithm, we will evaluate the learned HTN methods and the learned action models separately. First, it is possible that our `HTNLearn` learns more decomposition methods than that in its corresponding ground-truth model. We define two metrics to measure the learning accuracy, which are called *precision* and *recall*, respectively. *Precision* and *recall* are often used in Information Retrieval (IR) [58]. In our definition, we will take the similar meaning as used in IR. We denote the sets of all the learned method structures and the ground-truth method structures by S_l and S_g, respectively. Thus, we define *precision* and *recall* as follows (note that a learned structure and a ground-truth structure are the same if and only if both their

[8] http://planning.cis.strath.ac.uk/competition/

corresponding tasks and subtask lists are the same):

$$precision = \frac{S_l \cap S_g}{S_l},$$

and

$$recall = \frac{S_l \cap S_g}{S_g}.$$

Precision measures how many of the learned structures are actually ground-truth structures while *recall* measures how many of the ground-truth structures are actually learned. A perfect *precision* (i.e., a score of 1.0) means that every method learned by `HTNLearn` matches ground-truth method structures (but says nothing about whether all ground-truth structures were learned), whereas a perfect *recall* (i.e., a score of 1.0) means that all ground-truth structures were learned by `HTNLearn` (but says nothing about how well they match those structures).

We define the accuracy of the learned structures, denoted by Acc_s, as the combination of *precision* and *recall* (which can be viewed as the *F-measure* defined by [58]). Formally, Acc_s can be calculated by the following equation when neither *precision* nor *recall* is 0:

$$Acc_s = \frac{2}{\frac{1}{precision} + \frac{1}{recall}}$$

We define Acc_s to be 0 if either *precision* or *recall* is 0. Since the harmonic mean of *precision* and *recall* tends toward the smaller one, it tends to mitigate the impact of the larger one and aggravate the impact of the smaller one.

We would like to see how much human effort is needed to tune the applicability conditions of HTN models, which include *precondition* lists, *add* lists and *delete* lists of action models, and *precondition* lists of HTN methods. The human effort is measured by counting how many predicates need to be added to or deleted from the learned models to match the ground-truth models. Specifically, for each learned action model a, we denote its corresponding ground-truth action model as a_g. We define the error rate of a's preconditions as

$$E_{pre}(a) = \frac{(\text{PRE}(a) \cup \text{PRE}(a_g)) - (\text{PRE}(a) \cap \text{PRE}(a_g))}{allPossiblePre(a)}$$

where $allPossiblePre(a)$ means the set of all the possible preconditions of a (i.e., the set of all predicates in P whose parameters are included in a). Likewise, we can define the error rates of a's effects and a's effects, which is denoted as $E_{add}(a)$ and $E_{del}(a)$, respectively. For each learned method m whose structure $\text{STR}(m)$ is in $S_l \cap S_g$, we can calculate the error rate of m's preconditions accordingly, which is denoted by $E_{pre}(m)$. Furthermore, we define the error rate of the learned applicability conditions as the average of these four kinds of error rates, i.e.,

$$E_c = \frac{\sum_{a \in A}(E_{pre}(a) + E_{add}(a) + E_{del}(a)) + \sum_{\{m|\text{STR}(m) \in S_l \cap S_g\}} E_{pre}(m)}{3 * |A| + |S_l \cap S_g|},$$

and the accuracy as $Acc_c = 1 - E_c$. Note that when calculating error rates of preconditions of the learned methods, we only focus on methods whose method structures

belong to ground-truth structures S_g. Any other method is viewed as incorrect and do not need to be considered.

We will show experimental results with the accuracy measures Acc_s and Acc_c when complete decomposition trees are provided for the traces and when partial decomposition trees are provided (including the case where the partial tree is empty). We will also measure the running time of our HTNLearn with respect to different number of decomposition trees.

5.3. Experimental results

5.3.1. Accuracy with complete decomposition trees

First, we would like to see how the accuracy changes when increasing number of decomposition trees are given for the case when these decomposition trees are complete. In this setting, our learning problem is reduced to learning action models and method preconditions simultaneously, with no need to learn method structures (since method structures can be directly extracted from decomposition trees), as is done by our previous work [69].

An alternative to HTNLearn also solving this problem would be to learn the action models with ARMS [68] and separately learn the method preconditions with an existing algorithm such as CaMeL [29]. To determine the importance of learning the action models and method preconditions simultaneously, we ran an experiment comparing HTNLearn 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.

We varied the fraction of intermediate states provided from 1/5 to 1. The factor N/5 (with $1 \leq N \leq 5$) means only N intermediate states are observed among five sequential states. For the factor 1 (i.e., N = 5), all 5 intermediate states are observed. Note that each observed intermediate state is assumed to be *complete* (i.e., no atoms are missing in its description). We run this experiment five times and calculate the average accuracy. The results are shown in Figure 5. In all cases, the accuracies of HTNLearn are higher than that of ARMS$^+$. This is 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 when the two are learned separately. An increase in the number of intermediate states that are specified generally increases the accuracy of both systems.

We would like to test how accuracy changes when each observed intermediate state is not *complete*. We set the percentage of intermediate states as 1/3, and vary the percentage of propositions in each state from 20% to 100%. We run HTNLearn five times and calculate an average of accuracies. The result is shown in Figure 6. From Figure 6, we can see that the accuracy Acc_c generally increases when the percentage of observed propositions becomes higher. This is consistent with our intuition, since the more propositions are observed, the more knowledge can be used for improving the learning result. By comparing the curves of HTNLearn and ARMS$^+$, we can also find that the accuracy Acc_c of HTNLearn is higher than that of ARMS$^+$. The potential

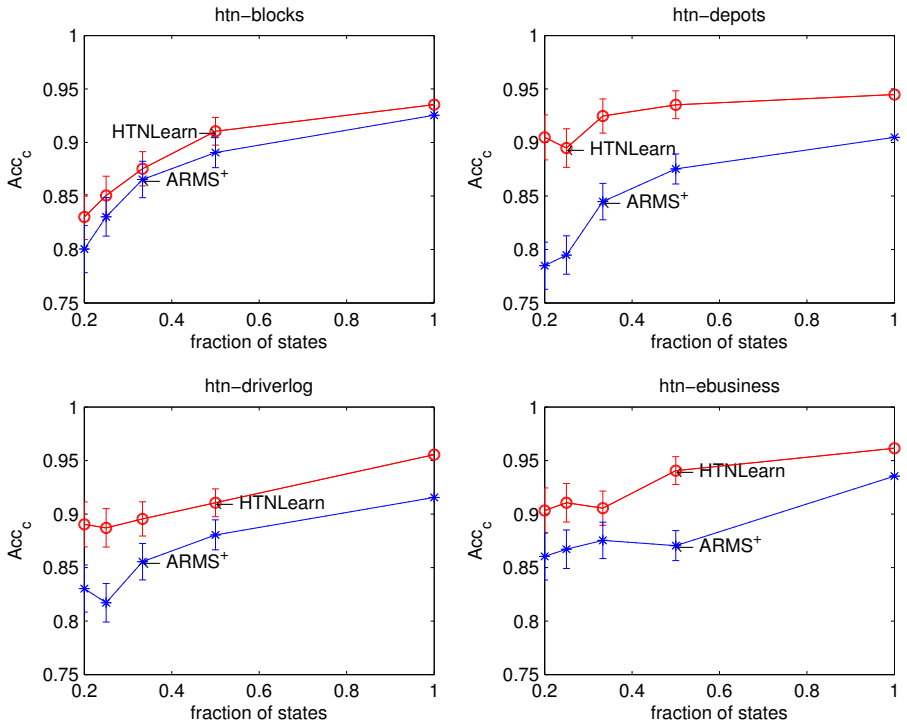


Figure 5: The accuracy of applicability conditions Acc_c with respect to the fraction of states, when provided with complete decomposition trees

reason is as stated in the previous paragraph, i.e., constraints on action models may help learn more accurate method preconditions.

5.3.2. Accuracy with partial decomposition trees

In this experiment, we would like to evaluate the performance of `HTNLearn` on learning method structures with only partial decomposition structures as input. We set the percentage of observed intermediate states to be 1/3; when provided, each of the intermediate states was complete. We varied the percentage of partial structures from 0, 25%, 50%, 75% to 100%. When the percentage is "0", it means that the partial decomposition tree is empty and only the plan trace is provided, while 100% means that the complete decomposition tree is provided. When the percentage is set to be 25%, it means there is one non-primitive task whose decomposition is known among four non-primitive tasks in each decomposition tree, likewise for other percentages. The results were shown in Figure 7.

From Figure 7, we can see that for all cases, when percentage of partial structures increases, Acc_s increases correspondingly. This is reasonable, since the larger the number of partial structures, the more correct structures can be extracted directly from the

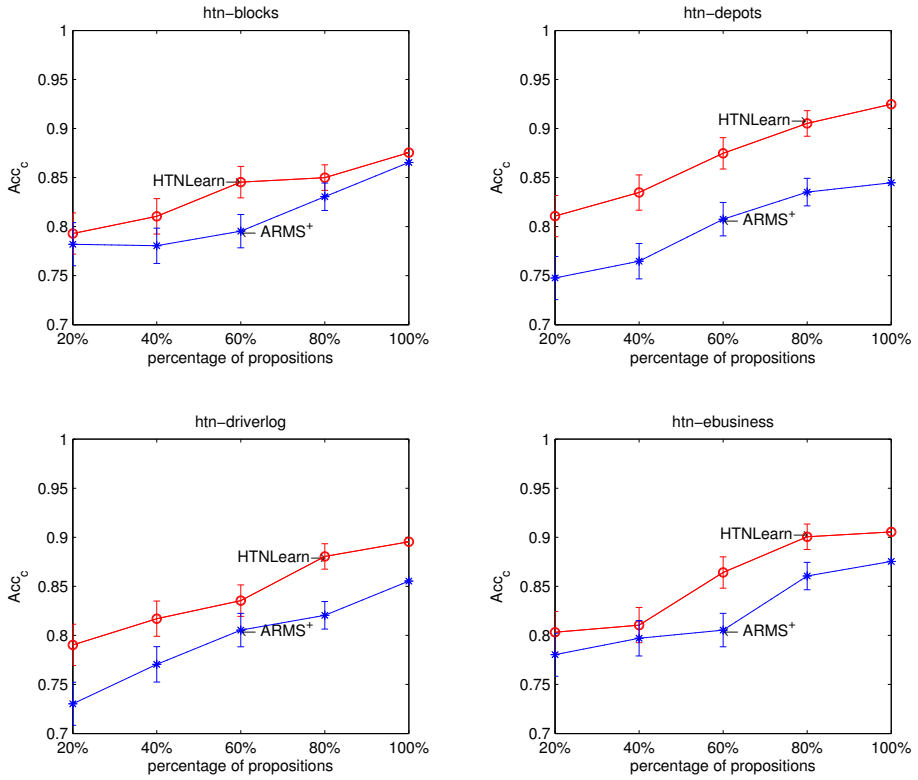


Figure 6: The accuracy of applicability conditions Acc_c with respect to the percentage of propositions, provided with complete decomposition trees

inputted partial decomposition trees. In addition, Acc_c also becomes higher when the percentage gets larger. This is because the structure information would provide valuable information for building decomposition constraints DC, which may help learn the HTN conditions. We observed that the accuracy of the learned structures Acc_s was not less than 0.8 for all the cases when the percentage was not less than 50%. This reveals our learning algorithm `HTNLearn` could indeed help acquire structure knowledge, which could be provided for people to build HTN models. We further test the human efforts saved by `HTNLearn` in Section 5.3.3. The number of method structures learned by `HTNLearn` is not large. To empirically show this fact, we recorded the average numbers (we ran `HTNLearn` five times to calculate an average) of method structures learned for all domains by setting the percentage of partial structures to be 50%. The results were 13, 15, 18, and 7 method structures learned for the *htn-blocks*, *htn-driverlog*, *htn-depots*, and *htn-business* domains, respectively.

We now consider the relations between `HTNLearn` and other learning systems.

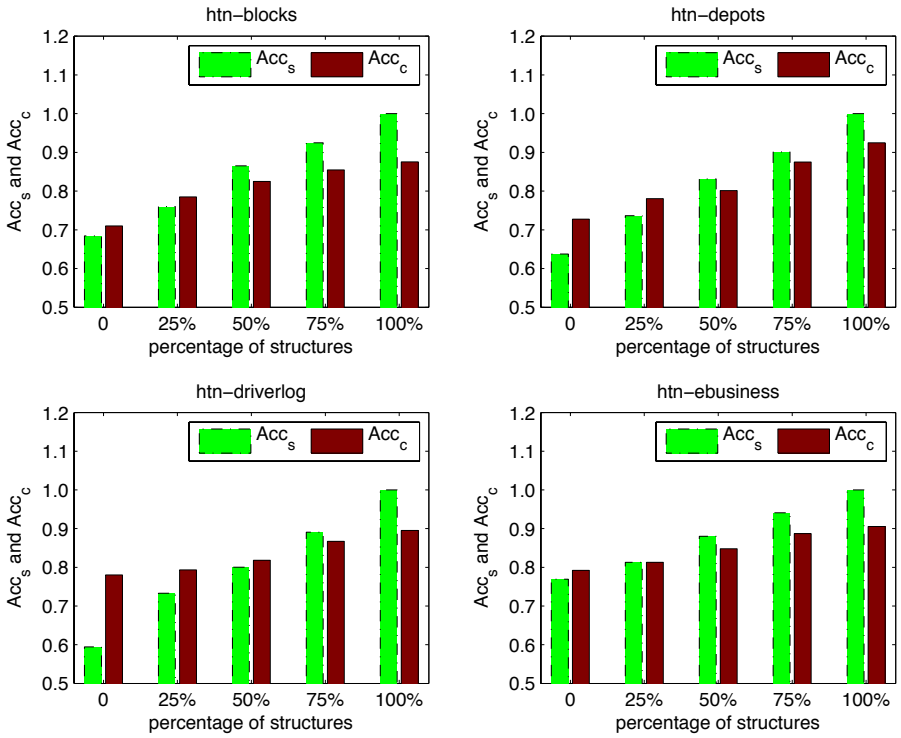


Figure 7: The accuracies of Acc_s and Acc_c with respect to different percentages of structures in partial decomposition trees

There are no HTN learning systems that can be trivially extended to learn HTN structures without knowing action models. Extending some techniques, such as grammar inferencing techniques, to first learn HTN structures and then action models neglects the close relations between HTN structures and action models. These relations are helpful for improving the learning quality of HTN models as demonstrated in our experiments. In particular, action models can be used to generate intermediate states between actions by executing plan traces. These states can be further used to recognize HTN structures with the help of annotated tasks (i.e., which help determine what action subsequence achieves which annotated task by checking if preconditions and effects of the annotated task are satisfied in the first and last state of the subsequence respectively). In other words, action models are helpful for learning HTN structure. Conversely, the improved HTN structures can also help learn action models, since HTN structures suggest close relations between actions (e.g., actions produces conditions for their subsequent actions). These relations are useful for learning preconditions and effects of actions.

We would like to examine any relationship between Acc_s and Acc_c. To do so, we fit curves to see how Acc_c changes when Acc_s becomes larger. The results are shown

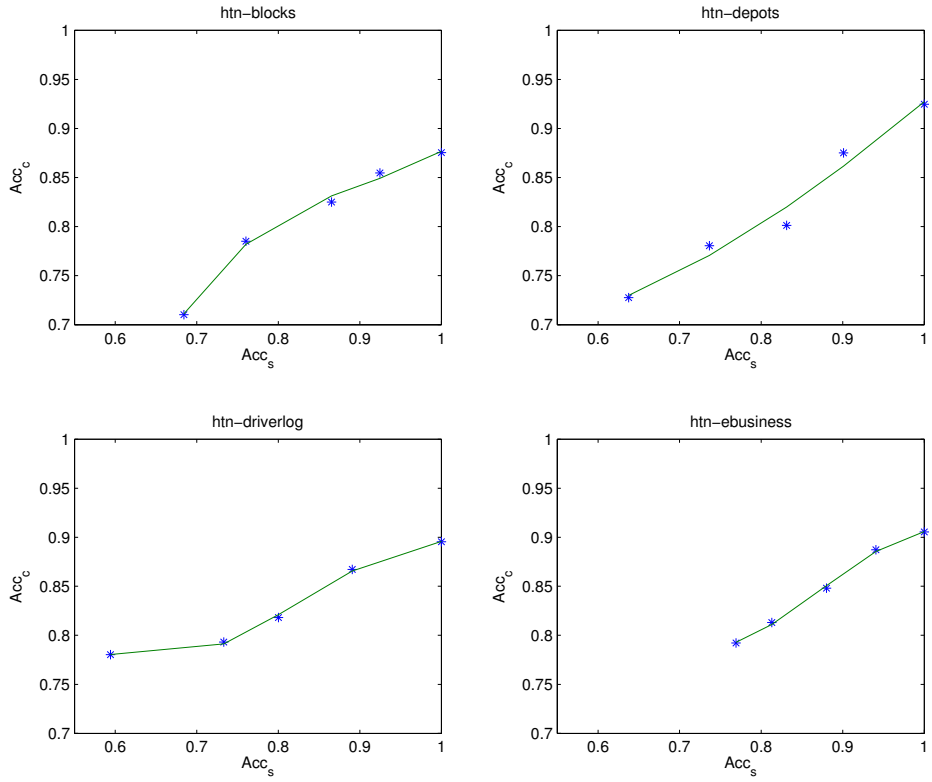


Figure 8: Accuracy Acc_c changes with respect to different Acc_s

in Figure 8. We find that Acc_c becomes larger as Acc_s gets larger for all domains. This exhibits that structure information could really help learn HTN conditions. The potential reason could be the same mentioned in the previous paragraph, i.e., structure information may help constrain the relationship between conditions, which may, as a result, help improve the learning result of the HTN conditions.

5.3.3. Human efforts saved by `HTNLearn`

In this experiment, we would like to see how much benefit can be attained by `HTNLearn` when creating new HTN models, i.e., how much effort can be saved by `HTNLearn`. To do this, we invited two groups of people to do testing, each of which had eight members. We took care to balance members of these groups; in the first group, six were university students and faculty, and two were engineers from a company. In the second group, five were university students and faculty and the remaining three were engineers from a company. All participants were between 20 and 35 years of age, and had some general background knowledge about AI planning. All partic-

ipants were told the description of domains, including the meaning of each task and each action, in natural language (i.e., without any logic-language description).

We conducted two user studies with the above-mentioned two groups of people. In the first case, we ran HTNLearn by setting the percentage of states to be 1/3 (each of which was complete), the percentage of partial structures to be 50%. The first group are told to create HTN models based on the learned models. In the second case, we let the second group create HTN models without any learned HTN models. We recorded the times used to create HTN models by these two groups, and calculated their accuracies (including Acc_s and Acc_c), respectively. The results are shown in Figures 9 and 10, where the red "◊" indicates the testing result of the first case (i.e., building HTN models based on learned HTN models) and the blue "□" indicates the testing result of the second case (i.e., building HTN models without learned HTN models).

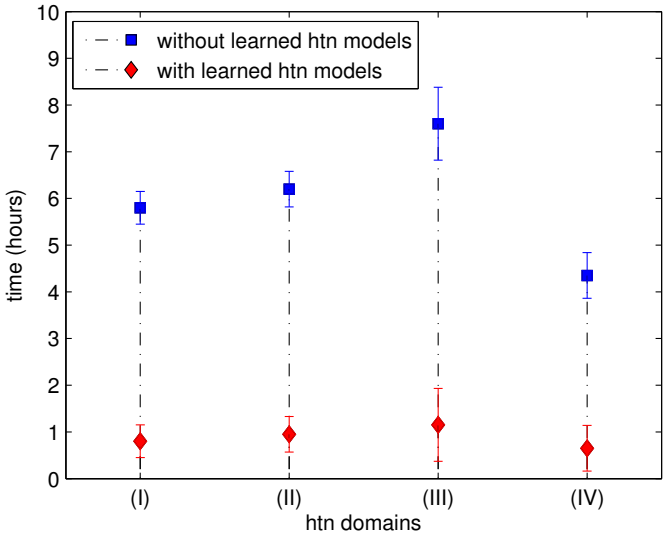


Figure 9: Time taken when creating HTN models with or without the learnt HTN models, where (I)-(IV) are HTN domains *htn-blocks*, *htn-driverlog*, *htn-depots*, and *htn-ebusiness*, respectively.

From Figure 9, we found that the time spent in the first case was much lower than that of the second case for all four HTN domains. This exhibits that our HTNLearn algorithm can indeed save time in creating HTN models. Furthermore, we found that the accuracies of Acc_s and Acc_c of the first case were much higher than those of the second case. This indicates that HTNLearn can indeed improve the model quality when creating HTN models.

In summary, from Figure 7 we can see that, when the percentage of structures is set to be 50%, the accuracies of structures and conditions are both not less than 80%. Furthermore, from Figures 9 and 10 show the learnt HTN models can indeed largely help humans create more accurate models, as well as saving times in creating HTN models, when the percentage of a structure is set to be 50%. Based on these

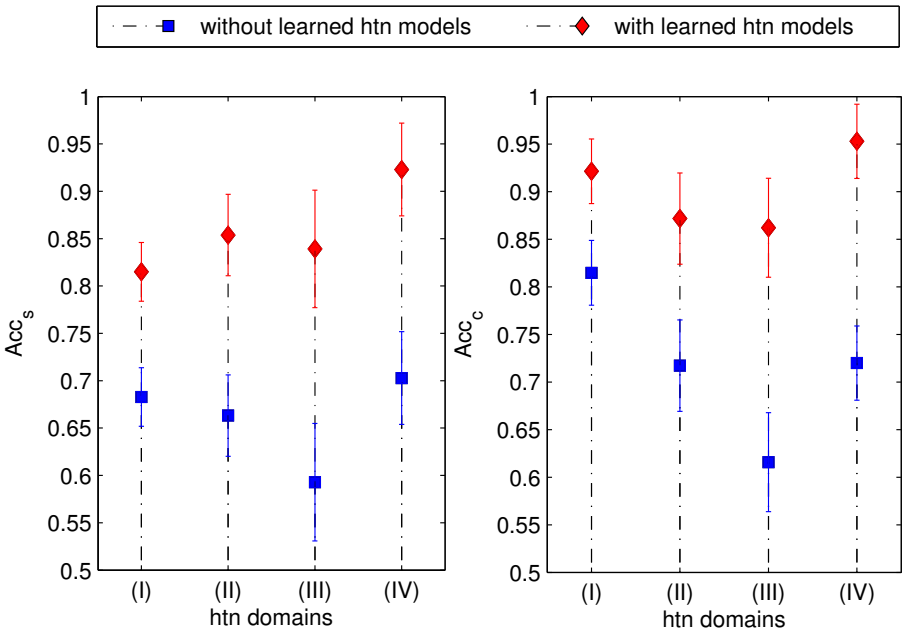


Figure 10: Accuracies Acc_c and Acc_s attained when creating HTN models with or without the learnt HTN models, where (I)-(IV) are HTN domains *htn-blocks*, *htn-driverlog*, *htn-depots*, and *htn-ebusiness*, respectively.

observations, we conjecture that learning with a structure percentage larger than 50% might produce acceptably accurate models for humans to further refine and finally attain high-quality models.

5.3.4. Running time

To test the running time of `HTNLearn`, we ran `HTNLearn` with respect to different percentages of partial structures, by setting the percentage of states to be 1/3, the number of partial decomposition trees to be 200, and the percentage of structures in the partial decomposition trees was set to 50%. We randomly selected the states (with a probability of 1/3) five times, and recorded an average of the running time. The results are shown in Figure 11. We can see that in all cases the running time generally goes down as the percentage of structures becomes higher. The reason for this is because the problem space (or the number of clauses or variables of the MAX-SAT problem) becomes smaller as the percentage of structures becomes larger, i.e., there is no need to search a large number of possible *method structures*. This reduces the running time for the MAX-SAT solver to get the final solution. To verify the fact that the problem space becomes smaller as the percentage of structures becomes larger, we show the

change of the number of clauses and variables of the MAX-SAT problem with respect to the percentage of structures in domain *htn-blocks* in Table 4. From Table 4, we can see that the number of clauses or variables generally decreases when the percentage of structures becomes larger, which indicates the problem space becomes smaller with the percentage increasing in the *htn-blocks* domain (and likewise for other domains).

Table 4: The change of clauses and variables w.r.t. percentage of structures in domain *htn-blocks*.

percentage of structures	number of clauses	number of variables
0%	12893	812
25%	9358	695
50%	5343	632
75%	3841	586
100%	2356	528

We also would like to see how the running time changes when increasing number of decomposition trees are given. We also set the percentage of states to be 1/3, each of which was complete, and the percentage of partial structures to be 50%. The result is shown in Table 5. The running time of `HTNLearn` increases polynomially with the size of the input. To verify our claim, we use the relationship between the size of the given partial decomposition trees and the CPU 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. The fitting curve for the domain *htn-blocks* can be found in Figure 12 (other curves for other domains that are not shown here are similar to Figure 12). We computed the polynomial for fitting *htn-blocks*, which is $-0.0002x^3 + 0.0527x^2 - 2.0466x + 29.7333$.

Table 5: Column 1 is the number of partial decomposition trees, columns 2-5 are cpu times (seconds)

number	htn-blocks	htn-driverlog	htn-depots	htn-ebusiness
20	9	10	15	6
40	28	32	46	23
60	51	63	76	36
80	122	152	189	116
100	214	221	265	182
120	282	306	299	214
140	346	365	389	327
160	422	413	499	393
180	481	501	543	433
200	504	583	593	471

5.4. Summary

From the experiments we draw following conclusions:

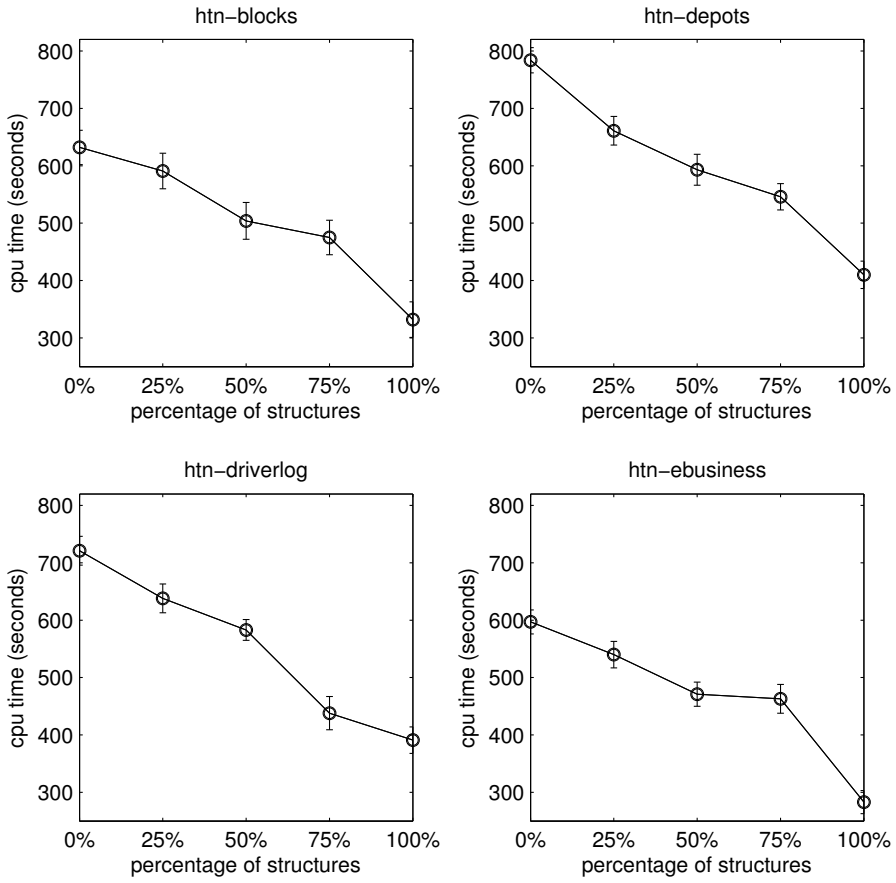


Figure 11: The running time with respect to different percentages of structures.

- Our `HTNLearn` algorithm can effectively learn HTN models, including structures and conditions, from partially observed plan traces with high accuracy. The accuracy of the leaned models increases the more plan traces are provided.
- Our `HTNLearn` algorithm can help reduce the human effort in creating HTN models. Comparing to creating HTN models from scratch, manually extending the learned models saves time and results in more accurate models.
- Our `HTNLearn` algorithm requires a small amount of training time to learn high-quality models; its running time increases polynomially on the number of plan traces given.

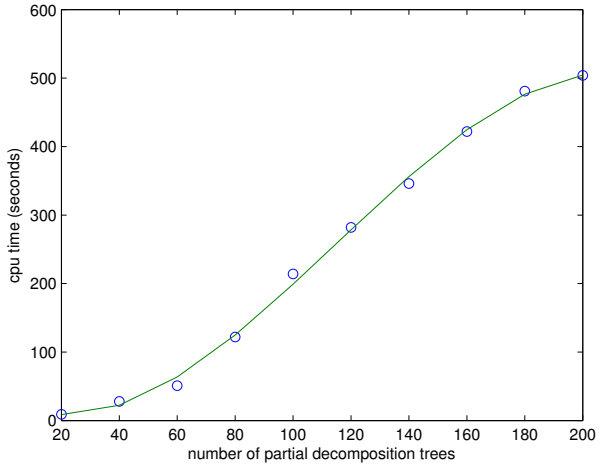


Figure 12: Fitted curve of the relationship between CPU time and number of partial decomposition trees for *htn-blocks*.

6. Final remarks

In this paper, we presented a novel algorithm, `HTNLearn`, to learn the action models and method preconditions of an HTN model. Given a set of plan traces annotated with partial state information and possibly annotated with partial decomposition trees, `HTNLearn` builds a set of state, decomposition, and action weighted constraints, and solves them with a MAX-SAT solver. The solution obtained is the HTN model that best explains the observed input information.

To the best of our knowledge, `HTNLearn` is the first algorithm capable of learning all parts of an HTN model, namely, (1) the collection of methods, each indicating the task, subtasks, and preconditions and (2) the collection of action models, each indicating the preconditions and effects.

From the experimental results, we conclude that our `HTNLearn` algorithm can learn initial HTN models that are fairly accurate and that this accuracy increases with the number of plan traces given. These initial models can be given to humans which need to perform some, comparatively, minor editing effort to attain a complete HTN domain.

There are a number of potential future directions that can be pursued. First, we note from the experimental results, that when given complete decomposition trees, `HTNLearn` is able to learn a completely accurate model of the HTN structure. However, this is not the case for the applicability conditions; even when completely observable states and complete decomposition trees are provided, `HTNLearn` is unable to learn completely accurate applicability conditions. This is a consequence of two factors: (1) the constraints represent likely explanations for situations observed in the traces. (2) The MAXSAT solver we use will yield a solution that covers most of the constraints but will not guarantee that it will solve all of them. These two factors

are what enables HTNLearn to elicit highly accurate HTN models under partial state observability. Other HTN learning algorithms such as HTN-MAKER guarantee that correct conditions are learned but assume complete state observability. An interesting research direction is to automatically identify when complete traces are given or when the HTN model is converging towards a complete domain, in which case a different learning mechanism can be triggered. Another interesting direction is to make HTNLearn incremental; HTNLearn elicits the model from a set of traces. If new traces are provided, HTNLearn will need to be ran again with the old and new traces. It will be interesting to refine the HTN model elicited so far rather than constructing a new one from scratch. It could also be interesting to explore more constraints to improve the learning quality, e.g., exploring constraints on the likelihood of a predicate in a state being a precondition of all subsequent actions after that state. Furthermore, it is also interesting to study the feasibility of refining automatically the initial HTN model produced by HTNLearn, by executing plans generated by our initial model in a simulator and refine the model based on the feedback we observe from the simulation.

Perhaps the most important and more difficult future research direction is the learning of HTN models that can reason with more complex representations. This is an important requirement for real-world applicability of this line of research. One possibility will be learning HTN models that reason with numerical information. For example, in the web service composition accessing some information sources might have associated costs and plans are to be built under budgetary constraints. Another possibility is learning complex applicability conditions such as atoms inferred from Horn clauses, which is a staple capability in SHOP and SHOP2 that enables to manually create complex domains.

Acknowledgments

Hankz Hankui Zhuo's research is supported by the National Natural Science Foundation of China (No. 61309011). Héctor Muñoz-Avila's work is supported in part by NSF grant 0642882. Qiang Yang thanks China National 973 project 2014CB340304 for kind support. Qiang Yang is also supported by Hong Kong RGC grants 621011 and 620812.

References

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

[2] Anderson, K. M., Sherba, S. A., Lepthien, W. V., 2002. Towards largescale information integration. In: Proceedings of International Conference on Software Engineering (ICSE-02).

[3] Asuncion, H. U., Asuncion, A. U., Taylor, R. N., 2010. Software traceability with topic modeling. In: Proceedings of International Conference on Software Engineering (ICSE-10).

[4] Baum, J., Nicholson, A. E., Dix, T. I., 2012. Proximity-based non-uniform abstractions for approximate planning. Journal of Artificial Intelligence Research 43, 477–522.

[5] Benson, S., 1995. Inductive learning of reactive action models. In: Proceedings of the Twelfth International Conference on Machine Learning (ICML-95). pp. 47–54.

[6] Blythe, J., Kim, J., Ramachandran, S., Gil, Y., 2001. An integrated environment for knowledge acquisition. In: Proceedings of IUI. pp. 13–20.

[7] Borchers, B., Furman, J., 1998. A two-phase exact algorithm for MAX-SAT and weighted MAX-SAT problems. J. Comb. Optim. 2 (4), 299–306.

[8] Botea, A., Enzenberger, M., Muller, M., Schaeffer, J., 2005. Macro-ff: Improving ai planning with automatically learned macro-operators. Journal of Artificial Intelligence Research 24, 581–621.

[9] Bryant, R. E., 1992. Symbolic boolean manipulation with ordered binary-decision diagrams. ACM Computing Surveys 24 (3), 293–318.

[10] Chrisman, L., 1992. Abstract probabilistic modeling of action. In: Proceedings of the First International Conference on Artificial Intelligence Planning Systems (AIPS-92). pp. 28–36.

[11] Coles, A., Smith, A., 2007. Marvin: A heuristic search planner with online macro-action learning. Journal of Artificial Intelligence Research 28, 119–156.

[12] Cresswell, S., McCluskey, T. L., West, M. M., 2009. Acquisition of object-centred domain models from planning examples. In: Proceedings of the Nineteenth International Conference on Automated Planning and Scheduling (ICAPS-09).

[13] Currie, K., Tate, A., 1991. O-plan: The open planning architecture. Artificial Intelligence 52 (1), 49–86.

[14] Estlin, T. A., Mooney, R. J., 1996. Multi-strategy learning of search control for partial-order planning. In: Proceedings of the Thirteenth National Conference on Artificial Intelligence. pp. 843–848.

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

[16] Fikes, R. E., Hart, P. E., Niisson, N. J., 1972. Learning and executing generalized robot plans. Artificial Intelligence 3, 251–288.

[17] Gajos, K. Z., Weld, D. S., Wobbrock, J. O., 2008. Decision-theoretic user interface generation. In: Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence (AAAI-08). pp. 1532–1536.

[18] Garland, A., Ryall, K., Rich, C., 2001. Learning hierarchical task models by defining and refining examples. In: Proceedings of the First International Conference on Knowledge Capture (K-CAP). pp. 44–51.

[19] Ghallab, M., Nau, D., Traverso, P., 2004. Automated Planning: Theory and Practice. Morgan Kaufmann.

[20] Gil, Y., 1994. Learning by experimentation: Incremental refinement of incomplete planning domains. In: Proceedings of the Eleventh International Conference on Machine Learning (ICML-94). pp. 87–95.

[21] He, R., Brunskill, E., Brunskill, E., 2010. Puma: Planning under uncertainty with macro-actions. In: Proceedings of AAAI. pp. 1089–1095.

[22] Hernandez, T., Kambhampati, S., 2004. Integration of biological sources: Current systems and challenges ahead. SIGMOD Record 33 (3), 51–60.

[23] Hoffmann, J., Bertoli, P., Pistore, M., 2007. Web service composition as planning revised: in between background theories and initial state uncertainty. In: Proceedings of the Twenty-Second AAAI Conference on Artificial Intelligence (AAAI 2007). pp. 1013–1018.

[24] Hogg, C., Kuter, U., Muñoz-Avila, H., 2009. Learning hierarchical task networks for nondeterministic planning domains. In: Proceedings of IJCAI. pp. 1708–1714.

[25] Hogg, C., Kuter, U., Munoz-Avila, H., 2010. Learning methods to generate good plans: Integrating htn learning and reinforcement learning. In: Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence (AAAI-10).

[26] Hogg, C., Muñoz-Avila, H., Kuter, U., 2008. HTN-MAKER: Learning HTNs with minimal additional knowledge engineering required. In: Proceedings of AAAI. pp. 950–956.

[27] Holmes, M. P., Jr., C. L. I., 2004. Schema learning: Experience-based construction of predictive action models. In: In Advances in Neural Information Processing Systems 17 (NIPS-04).

[28] Iba, G. A., 1989. A heuristic approach to the discovery of macro-operators. Machine Learning 3, 285–317.

[29] Ilghami, O., Muñoz-Avila, H., Nau, D. S., Aha, D. W., 2005. Learning approximate preconditions for methods in hierarchical plans. In: Proceedings of ICML. pp. 337–344.

[30] Korf, R. E., 1985. Macro-operators: A weak method for learning. Artificial Intelligence 26, 35–77.

[31] Kuter, U., Nau, D., Pistore, M., Traverso, P., 2009. Task decomposition on abstract states, for planning under nondeterminism. Artificial Intelligence 173, 669–695.

[32] Kuter, U., Sirin, E., Parsia, B., Nau, D., Hendler, J., 2005. Information gathering during planning for web service composition. Journal of Web Semantics (JWS) 3 (2-3), 183–205.

[33] Lau, T., Domingos, P., Weld, D. S., 2000. Version space algebra and its application to programming by demonstration. In: In Proceeding of The Seventeenth International Conference on Machine Learning (ICML-00). pp. 527–534.

[34] LI, C. M., Manya, F., Mohamedou, N., Planes, J., 2009. Exploiting cycle structures in Max-SAT. In: Proceedings of 12th international conference on the Theory and Applications of Satisfiability Testing (SAT-09). pp. 467–480.

[35] LI, C. M., Manya, F., Planes, J., 2006. Detecting disjoint inconsistent subformulas for computing lower bounds for Max-SAT. In: Proceedings of the 21st National Conference on Artificial Intel ligence (AAAI-06). pp. 86–91.

[36] LI, C. M., Manya, F., Planes, J., October 2007. New inference rules for Max-SAT. Journal of Artificial Intelligence Research 30, 321–359.

[37] Li, N., Kambhampati, S., Yoon, S. W., 2009. Learning probabilistic hierarchical task networks to capture user preferences. In: Proceedings of IJCAI. pp. 754–759.

[38] Marthi, B., Russell, S., Wolfe, J., 2008. Angelic hierarchical planning: Optimal and online algorithms. In: Proceedings of ICAPS. pp. 222–231.

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

[40] Minton, S., 1990. Quantitative results concerning the utility of explanation-based learning. Artificial Intelligence 42 (2), 363–391.

[41] Minton, S., Carbonell, J. G., Knoblock, C. A., Kuokka, D. R., Etzioni, O., Gil, Y., 1989. Explanation-based learning: A problem solving perspective. Artificial Intelligence 40 (1), 63–118.

[42] Muggleton, S., Raedt, L. D., 1994. Inductive logic programming: Theory and methods. Journal of Logic Programming 19/20, 629–679.

[43] Nance, M., Vogel, A., Amir, E., 2006. Reasoning about partially observed actions. In: Proceedings of the Twenty-First National Conference on Artificial Intelligence (AAAI-06). pp. 888–893.

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

[45] Nau, D. S., Cao, Y., Lotem, A., Muñoz-Avila, H., 1999. SHOP: Simple hierachical ordered planner. In: Proceedings of IJCAI. pp. 968–973.

[46] Nejati, N., Langley, P., Konik, T., 2006. Learning hierarchical task networks by obervation. In: Proceedings of ICML. pp. 665–672.

[47] Newton, M. H., Levine, J., Fox, M., Long, D., 2007. Learning macro-actions for arbitrary planners and domains. In: Proceedings of ICAPS. pp. 256–263.

[48] Oates, T., Cohen, P. R., 1996. Searching for planning operators with context-dependent and probabilistic effects. In: Proceedings of the Thirteenth National Conference on Artificial Intelligence (AAAI-96). pp. 865–868.

[49] Pasula, H. M., Zettlemoyer, L. S., Kaelbling, L. P., 2004. Learning probabilistic relational planning rules. In: Proceedings of the Fourteenth International Conference on Automated Planning and Scheduling (ICAPS-04). pp. 73–82.

[50] Pasula, H. M., Zettlemoyer, L. S., Kaelbling, L. P., 2007. Learning symbolic models of stochastic domains. Journal of Artificial Intelligence Research 29, 309–352.

[51] Reddy, C., Tadepalli, P., 1997. Learning goal-decomposition rules using exercises. In: Proceedings of ICML. pp. 278–286.

[52] R.M.Simpson, McCluskey, T. L., Zhao, W., R.S.Aylett, Doniat, C., 2001. Gipo: An integrated graphical tool to support knowledge engineering in ai planning. In: Proceedings of the European Conference on Planning.

[53] Sablon, G., Bruynooghe, M., 1994. Using the event calculus to integrate planning and learning in an intelligent autonomous agent. In: In Current Trends in AI Planning. pp. 254–265.

[54] Schmill, M. D., Oates, T., Cohen, P. R., 2000. Learning planning operators in real-world, partially observable environments. In: Proceedings of the Fifth International Conference on Artificial Intelligence Planning Systems (AIPS-00). pp. 246–253.

[55] Shahaf, D., Amir, E., 2006. Learning partially observable action schemas. In: Proceedings of the Twenty-First National Conference on Artificial Intelligence (AAAI-06). pp. 913–919.

[56] Shahaf, D., Chang, A., Amir, E., 2006. Learning partially observable action models: Efficient algorithms. In: Proceedings of the Twenty-First National Conference on Artificial Intelligence (AAAI-06). pp. 920–926.

[57] Simpson, R. M., Kitchin, D. E., McCluskey, T. L., 2007. Planning domain definition using gipo. Knowledge Eng. Review 22 (2), 117–134.

[58] van Rijsbergen, C. J., 1979. Information Retrieval. Butterworth-Heinemann.

[59] Walsh, T. J., Littman, M. L., 2008. Efficient learning of action schemas and web-service descriptions. In: Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence (AAAI 2008). pp. 714–719.

[60] Wang, X., 1995. Learning by observation and practice: An incremental approach for planning operator acquisition. In: Proceedings of the Twelfth International Conference on Machine Learning (ICML-95). pp. 549–557.

[61] Wickler, G., Potter, S., Tate, A., Piechouicek, M., Semsch, E., 2007. Planning and choosing: Augmenting htn-based agents with mental attitudes. In: IAT. pp. 222–228.

[62] Wilkins, D. E., 1990. Can ai planners solve practical problems? Computational Intelligence 6, 232–246.

[63] Winner, E., Veloso, M., 2002. Analyzing plans with condition effects. In: Proceedings of the Sixth International Conference on AI Planning and Scheduling (AIPS-02).

[64] Wu, D., Parsia, B., Sirin, E., Hendler, J., , Nau, D., Nau, D., 2003. Automating DAML-S web services composition using shop2. In: Proceedings of 2nd International Semantic Web Conference (ISWC2003).

[65] Xu, K., Muñoz-Avila, H., 2005. A domain-independent system for case-based task decomposition without domain theories. In: Proceedings of AAAI. pp. 234–240.

[66] Yang, Q., 1990. Formalizing planning knowledge for hierarchical planning. Computational Intelligence Journal 6 (2), 12–24.

[67] Yang, Q., Wu, K., Jiang, Y., 2005. Learning actions models from plan examples with incomplete knowledge. In: Proceedings of the Fifteenth International Conference on Automated Planning and Scheduling (ICAPS-05). pp. 241–250.

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

[69] Zhuo, H. H., Hu, D. H., Hogg, C., Yang, Q., Muñoz-Avila, H., 2009. Learning HTN method preconditions and action models from partial observations. In: Proceedings of IJCAI. pp. 1804–1810.

[70] Zhuo, H. H., Yang, Q., Hu, D. H., Li, L., 2010. Learning complex action models with quantifiers and logical implications. Artificial Intelligence Journal 174 (18), 1540–1569.