# HTN-MAKER: Learning HTNs with Minimal Additional Knowledge Engineering Required

**Chad Hogg** and **Héctor Muñoz-Avila**
Department of Computer Science & Engineering
Lehigh University
Bethlehem, Pennsylvania 18015, USA

**Ugur Kuter**
University of Maryland,
Institute for Advanced Computer Studies,
College Park, Maryland 20742, USA

**Abstract**

We describe HTN-MAKER, an algorithm for learning hierarchical planning knowledge in the form of decomposition methods for Hierarchical Task Networks (HTNs). HTN-MAKER takes as input the initial states from a set of classical planning problems in a planning domain and solutions to those problems, as well as a set of semantically-annotated tasks to be accomplished. The algorithm analyzes this semantic information in order to determine which portions of the input plans accomplish a particular task and constructs HTN methods based on those analyses.

Our theoretical results show that HTN-MAKER is sound and complete. We also present a formalism for a class of planning problems that are more expressive than classical planning. These planning problems can be represented as HTN planning problems. We show that the methods learned by HTN-MAKER enable an HTN planner to solve those problems. Our experiments confirm the theoretical results and demonstrate convergence in three well-known planning domains toward a set of HTN methods that can be used to solve nearly any problem expressible as a classical planning problem in that domain, relative to a set of goals.

## Introduction

A key challenge of automated planning is the requirement of a domain expert to provide some sort of background planning knowledge about the dynamics of the planning domain. At a minimum, classical planners require semantic descriptions (i.e., preconditions and effects) of possible actions. More recent planning paradigms allow or require the expert to provide additional knowledge about the structural properties of the domain and problem-solving strategies. In many realistic planning domains, however, additional planning knowledge may not be completely available; this is partly because it is very difficult for the experts to compile such knowledge due to the complexities in the domains and it is partly because there is limited access to an expert to provide it. Thus, it is crucial to develop learning techniques in order to produce planning knowledge when human contributions are limited or unavailable.

One of the best-known approaches for modeling planning knowledge about a problem domain is ***Hierarchical Task Networks (HTNs)***. An HTN planner formulates a plan via ***task-decomposition methods*** (also known as ***HTN methods***), which describe how to decompose complex tasks (i.e., symbolic representations of activities to be performed) into simpler subtasks until tasks are reached that correspond to actions that can be performed directly. The basic idea was developed in the mid-70s (Sacerdoti 1975), and the formal underpinnings were developed in the mid-90s (Erol, Hendler, and Nau 1996). More recently, the HTN planner SHOP (Nau et al. 1999) has demonstrated impressive speed gains over earlier classical planners by using HTN-based domain-specific strategies for problem-solving while performing domain-independent search. HTNs provide a natural modeling framework in many real-world applications including evacuation planning, manufacturing, and UAV management planning (Nau et al. 2005).

In this paper, we describe a new technique to learn HTN methods. Our contributions are as follows:

- We describe HTN-MAKER (Hierarchical Task Networks with Minimal Additional Knowledge Engineering Required), an offline and incremental algorithm for learning HTN methods. HTN-MAKER receives as input a set of planning operators, a collection of initial states from classical planning problems and solutions to those problems, and a collection of annotated tasks (defined below). It produces a set of HTN methods. Together with the given operators, the learned HTN methods can be used by an HTN planner such as SHOP to solve HTN planning problems.

- We present theoretical results showing that the HTN-MAKER algorithm is sound and complete with respect to a set of goals. We formalize a class of planning problems that are more expressive than classical planning and can be represented as HTN planning problems. Our theoretical results show that the methods learned by HTN-MAKER enable an HTN planner to solve planning problems in that class.

- We demonstrate through experiments in three well-known planning domains, namely Logistics, Blocks-World, and Satellite, that HTN-MAKER returns a set of HTN methods capable of solving new planning problems on which it was not trained.

## Preliminaries

We use the usual definitions for HTN planning as in Chapter 11 of (Ghallab, Nau, and Traverso 2004). A ***state*** $s$ is a collection of ground atoms. A ***planning operator*** is a 4-tuple $o = (h, \mathsf{Pre}, \mathsf{Del}, \mathsf{Add})$, where $h$ (the ***head*** of the operator) is a logical expression of the form $(n\ arg_1\ \ldots\ arg_l)$ such that $n$ is a symbol denoting the name of the operator and each argument $arg_i$ is either a logical variable or constant symbol. The ***preconditions***, ***delete list*** and ***add list*** of the planning operator, $\mathsf{Pre}$, $\mathsf{Del}$, and $\mathsf{Add}$ respectively, are logical formulas over literals.

An ***action*** $a$ is a ground instance of a planning operator. An action is ***applicable*** to a state $s$ if its preconditions hold in that state. The result of applying $a = (h, \mathsf{Pre}, \mathsf{Del}, \mathsf{Add})$ to $s$ is a new state $s' = \mathsf{APPLY}(s, a) = (s \setminus \mathsf{Del}) \cup \mathsf{Add}$. A ***plan*** $\pi$ is a sequence of actions.

A ***task*** is a symbolic representation of an activity in the world, formalized as an expression of the form $(t\ arg_1\ \ldots\ arg_q)$ where $t$ is a symbol denoting the name of the activity and each $arg_i$ is either a variable or a constant symbol. A task can be either ***primitive*** or ***nonprimitive***. A primitive task corresponds to the head of a planning operator and denotes an action that can be directly executed in the world. A nonprimitive task cannot be directly executed; instead, it needs to be decomposed into simpler tasks until primitive ones are reached.

In this paper, we restrict ourselves to the Ordered Task Decomposition formalism of HTN planning (Nau et al. 1999). In this formalism, an ***HTN method*** is a procedure that describes how to decompose nonprimitive tasks into simpler ones. Formally, a ***method*** is a triple $m = (h, \mathsf{Pre}, \mathsf{Subtasks})$, where $h$ is a nonprimitive task (the ***head*** of the method), $\mathsf{Pre}$ is a logical formula denoting the ***preconditions*** of the method, and $\mathsf{Subtasks}$ is a totally-ordered sequence of ***subtasks***. A method $m$ is ***applicable*** to a state $s$ and task $t$ if the head $h$ of the method matches $t$ and the preconditions of the method are satisfied in $s$. The result of applying a method $m$ to a state $s$ and task $t$ is the state $s$ and sequence of $\mathsf{Subtasks}$.

An ***HTN planning problem*** is a 4-tuple $P^h = (s_0, T, O, M)$, where $s_0$ is the initial state, $T$ is the initial sequence of tasks, and $O$ and $M$ are sets of planning operators and methods respectively. A ***solution for the HTN planning problem*** $P^h$ is a plan (i.e., a sequence of actions) $\pi$ that, when executed in the initial state, performs the desired initial tasks $T$.

We define a ***classical planning problem*** as a 3-tuple $P^c = (s_0, g, O)$, where $s_0$ is the initial state, $g$ is the ***goals*** represented as a conjunction of logical atoms, and $O$ is a set of planning operators defined as above. A ***solution for the classical planning problem*** $P^c$ is a plan $\pi = \langle a_1, \ldots, a_k \rangle$ such that the state $s' = \mathsf{APPLY}(\mathsf{APPLY}(\ldots, (\mathsf{APPLY}(s_0, a_1), a_2), \ldots), a_k)$ satisfies the goals $g$.

In HTN planning, a task is simply a statement in predicate logic, with no semantics other than those provided by the methods that decompose it. We define an ***annotated task*** as a triple $t = (n, \mathsf{Pre}, \mathsf{Effects})$ where $n$ is a task, $\mathsf{Pre}$ is a set of atoms known as the preconditions, and $\mathsf{Effects}$ is a set of atoms known as the ***effects***. See the discussion of Figure 1 below for an example of an annotated task.

**Algorithm 1** HTN-MAKER$(P^c, \pi, \tau, M)$

```
1: Input: An initial state $s_0$ and a solution $\pi = \langle a_0, a_1, \ldots, a_k \rangle$
   for a classical planning problem $P^c$, a set of annotated tasks $\tau$,
   and a set of HTN methods $M$
2: Output: An updated set $M'$ of HTN methods
3: initialize $S \leftarrow (s_0)$
4: for $i \leftarrow 1$ to $k$ do
5:     $s_i \leftarrow \mathsf{APPLY}(s_{i-1}, a_{i-1})$ and append $s_i$ to $S$
6: initialize $X \leftarrow \emptyset$
7: for $f \leftarrow 1$ to $k$ do
8:     for $i \leftarrow f - 1$ down to 0 do
9:         for all $t = (n, \mathsf{Pre}, \mathsf{Effects})$ in $\tau$ do
10:            if $\mathsf{Effects} \subseteq s_f$ and $\mathsf{Effects} \not\subseteq s_i$ and $\mathsf{Pre} \subseteq s_i$ then
11:                $m \leftarrow \mathsf{LEARN}(\pi, t, M, X, i, f)$
12:                insert the new method $m$ into $M$
13:                insert $(m, i, f, \mathsf{Effects})$, the method $m$, its starting
                   and ending indices, and the effects of its annotated
                   task, into $X$
14: return $M$
```

The preconditions and effects associated with an annotated task as above give semantics for accomplishing the task and enable us to define an equivalence between an annotated task and a set of goals, and thus, between a classical planning problem and an HTN planning problem. Given a set of goal atoms $g$, we define the ***equivalent annotated task*** as $t = (n, \emptyset, g)$ for some task $n$. Then, the ***HTN-equivalent planning problem*** to a classical planning problem $P^c = (s_0, g, O)$ is an HTN planning problem $P^h = (s_0, \{n\}, O, M)$ such that $M$ is a set of HTN methods. This notion of equivalence between HTN and classical planning problems is crucial for analyzing the formal properties of our learning algorithm described in the next section.

## HTN-MAKER

HTN-MAKER is an incremental learning algorithm that produces a knowledge base of HTN methods from an input solution plan to a classical planning problem and successively updates its knowledge base when presented with solutions to new classical planning problems in the same planning domain.

Algorithm 1 shows a high-level description of HTN-Maker's top-level learning procedure. For an input initial state and a solution plan $\pi$ for a classical planning problem, HTN-MAKER first generates a list $S$ of states by applying the actions in $\pi$ starting from the initial state $s_0$ (see Lines 3–5).

The algorithm then traverses the states in which the effects of an annotated task might become true (Line 7), the states from which the accomplishment of those effects might begin (Line 8), and the annotated tasks whose effects might have been accomplished over that interval (Line 9). This traversal order is chosen deliberately to provide the best opportunity for learned methods to subsume each other. During this traversal, if there is an annotated task whose effects match the final state $s_f$ and whose preconditions match the start state $s_i$ (Line 10), then HTN-MAKER regresses the effects

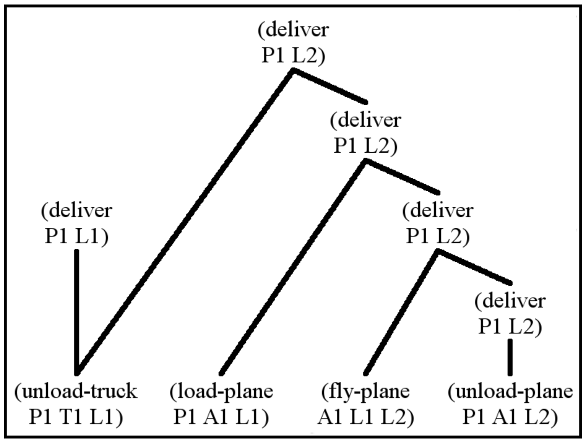

Figure 1: Example of HTN obtained by HTN-MAKER.

of the annotated task through the plan elements (actions in $\pi$ or methods learned previously) that caused those effects, in order to identify a sequence of subtasks that achieve the task and the preconditions necessary to ensure the success of those subtasks.

Unlike previous work on goal regression (Mitchell, Keller, and Kedar-Cabelli 1986), HTN-MAKER can regress goals both horizontally (through the actions) and vertically (up the task hierarchy through previously-learned methods). This is accomplished by the LEARN subroutine (Line 11, described below), which returns a new method for the annotated task with these subtasks and preconditions. HTN-MAKER then adds this method to the set of HTN methods learned so far (Line 12). HTN-MAKER also stores the instantiation of this method with the plan, along with the initial and final states indicating the subplan from which it was learned, so that in further processing it may be used as a subtask in a new method (Line 13). In Algorithm 1, the set $X$ is the storage for this purpose; it holds the method $m$, the indices of the starting and ending states in $S$ associated with $m$, and the effects of the annotated task associated with $m$.

Figure 1 demonstrates the inner workings of HTN-MAKER on a short plan in the Logistics domain from ICP-2. Suppose the input initial state consists of a package *P1* in a truck *T1* at an airport *L1* that contains an airplane *A1*. Suppose that we have a single annotated task in $\tau$, *(deliver ?p ?l)*, with preconditions that *?p* be a package and *?l* be a location, and effects that *?p* be at *?l*. After the first operator, the package *P1* has been delivered to location *L1*. Thus, HTN-MAKER learns a method for solving this task bound to these constants. The operator *(unload-truck P1 T1 L1)* produces the effect *(at P1 L1)*, so it will be selected as a subtask.

The learned method will be applicable when the types of variables are correct and the package is in a truck that is at the destination. In the next two states, there are no new valid instantiations of the task effects. In the final state, the package *P1* has been delivered to location *L2*, and at this time, HTN-MAKER learns a recursive series of methods for our delivery task as shown in Figure 1. The first method is for

**Algorithm 2** LEARN$(\pi, t, M, X, i, f)$

```
1: Input: A plan $\pi = \langle a_0, a_1, \ldots, a_n \rangle$, an annotated task $t = (n, \mathsf{Pre}, \mathsf{Effects})$, a set of HTN methods $M$, a set of method instantiations $X$, and initial and final state indices $i$ and $f$
2: Output: $m$ is a new method
3: $\mathsf{RemEff} \leftarrow \mathsf{Effects}$ ; $\mathsf{OutPre} \leftarrow \emptyset$ ; $c \leftarrow f$
4: $\mathsf{Subtasks} \leftarrow (\mathsf{verifiertask}(t))$
5: while $c > i$ do
6:   $Y = \{(m', k', c, \mathsf{Effects'}) \in X$ such that $\mathsf{Effects'} \cap (\mathsf{OutPre} \cup \mathsf{RemEff}) \neq \emptyset$ and $k' \geq i\}$
7:   if $Y \neq \emptyset$ then
8:     select $(m', k', c, \mathsf{Effects'}) \in X$ such that $m' = (n', \mathsf{Pre'}, \mathsf{Subtasks'}) \in M$
9:     prepend $n'$ to Subtasks
10:    $\mathsf{RemEff} \leftarrow \mathsf{RemEff} \setminus \mathsf{Effects'}$
11:    $\mathsf{OutPre} \leftarrow (\mathsf{OutPre} \setminus \mathsf{Effects'}) \cup \mathsf{Pre'}$
12:    $c \leftarrow k'$
13:  else
14:    if $a_{c-1} = (n', \mathsf{Pre'}, \mathsf{Del'}, \mathsf{Add'}) \in \pi$ and $\mathsf{Add'} \cap (\mathsf{OutPre} \cup \mathsf{RemEff}) \neq \emptyset$ then
15:      prepend $n'$ to Subtasks
16:      $\mathsf{RemEff} \leftarrow \mathsf{RemEff} \setminus \mathsf{Add'}$
17:      $\mathsf{OutPre} \leftarrow (\mathsf{OutPre} \setminus \mathsf{Add'}) \cup \mathsf{Pre'}$
18:    $c \leftarrow c - 1$
19: return $m = (n, \mathsf{Pre} \cup \mathsf{RemEff} \cup \mathsf{OutPre}, \mathsf{Subtasks})$
```

delivering a package that is in an airplane at the destination by unloading the airplane. The next delivers a package that is in an airplane at the wrong location by flying to the destination, which must be an airport, and then delivering it. The third requires that the package be at an airport that is not the destination and that contains an airplane, and proceeds by loading the package into airplane and then continuing to deliver. The final unloads from a truck, then continues from there to the final destination.

Algorithm 2 shows a high-level description of HTN-MAKER's LEARN subroutine. This subroutine first initializes the set of remaining effects RemEff, which represent the effects of the annotated task that are not accomplished by any subtask yet collected (Line 3). Next it initializes a set of outstanding preconditions OutPre (Line 3), a current state counter $c$ (Line 3), and a subtask list Subtasks (Line 4). The algorithm then collects a set $Y$ of method instantiations that (1) have an effect matching either a remaining effect or outstanding precondition, and (2) lie within the section of plan from which it is currently learning (Line 6). If this set is not empty (Line 7), it nondeterministically selects a member of the set to become a subtask (Line 8). Thus, its head is prepended to the subtask list Subtasks (Line 9), any remaining effects or outstanding preconditions that it accomplishes are removed (Lines 10–11), the preconditions of the selected subtask are added to the list of outstanding preconditions (Line 11), and the current state counter is moved to where the selected method instantiation started (Line 12).

If no appropriate method instantiation exists, then LEARN considers the operator that led to the current state as a possible subtask. The effects of the operator are checked against the remaining effects and outstanding preconditions (Line 14), and if they are found to be helpful then the operator is selected as a subtask. Thus the head of the operator is prepended to the subtask list (Line 15), the effects of the operator are removed from the remaining effects and outstanding preconditions (Lines 16–17), the preconditions of the operator are added to the outstanding preconditions (Line 17), and the current state counter is moved to the state from which the operator was instantiated (Line 18). If the operator is not helpful, it is skipped as irrelevant and the current state counter is adjusted to move before it (Line 18). This process continues until the current state counter reaches the initial state counter (Line 7). The LEARN subroutine returns a new method with the same head as the annotated task that was accomplished. The preconditions of this new method include any preconditions of the annotated task, any effects of the task that are not directly caused by any subtasks (the remaining effects), and any conditions necessary for the subtasks to be applicable (the outstanding preconditions); and its subtasks are the list that has been collected.

To guarantee soundness (see the next section), each annotated task has an associated verifier task (Line 6 of Algorithm 2). The HTN-MAKER algorithm creates a single method for the verifier task, which has the effects of the associated annotated task as its preconditions and no subtasks. (This is not shown in the high-level description in Algorithm 1.) This verifier is the last subtask of each learned method for the annotated task, ensuring that the method indeed accomplishes the effects of its associated annotated task.[1]

## Properties

In this section, we present the formal properties of the HTN-MAKER learning algorithm. The following theorem establishes the soundness of HTN-MAKER.

**Theorem 1.** *Let $O$ be a set of planning operators and $\tau$ be a set of annotated tasks for a classical planning domain. Suppose HTN-MAKER is given a set of classical planning problems and a solution plan for each of those planning problems, and it produces a set $M$ of HTN methods.*

*Then, for any classical planning problem $P^c$ in the domain, a solution $\pi$ produced by a sound HTN planning algorithm on the HTN-equivalent problem $P^h$ using the methods in $M$ will be a solution to $P^c$.*

The proof is omitted for lack of space. We provide an example demonstrating that without the verification tasks described previously, it is possible to reuse methods in unsound ways because a method selected for a subtask might result in the deletion of an atom that is needed by its parent (see Figure 2). Given the provided annotated tasks and a plan $\langle DE \rangle$, the first two methods could be learned, while the third would be learned from a plan $\langle F \rangle$. These methods could be used to perform the task decomposition shown in Figure 3. The annotated task states that "doAB" should have both "a" and "b" as effects, but only "b" has been accomplished.

[1]The verification subtasks in the learned methods may cause an HTN planner to frequently backtrack when it attempts to accomplish a task if the method does not produce the necessary effects. However, in our experiments on standard planning domains, we have never observed a single failure to decompose a verification task.

| | | | |
|---|---|---|---|
| Task doB | pre: | eff: b | |
| Task doAB | pre: | eff: a, b | |
| Op D | pre: | del: | add: a |
| Op E | pre: | del: | add: b |
| Op F | pre: a | del: a | add: b |
| Method doB | pre: | subtasks: E | |
| Method doAB | pre: | subtasks: D, doB | |
| Method doB | pre: a | subtasks: F | |

Figure 2: An example set of annotated tasks and planning operators in an artificial planning domain. The set of methods shown above are among the ones that would be learned by HTN-MAKER if verification subtasks were not used.

| State | Tasks |
|---|---|
| () | ( doAB ) |
| () | ( ( D ) ( doB ) ) |
| ( a ) | ( doB ) |
| ( a ) | ( F ) |
| ( b ) | () |

Figure 3: A possible decomposition of the annotated task doAB using the methods shown in Figure 2. Above, the decomposition generates a plan that violates the semantics of the task doAB because the plan, when executed, does not achieve the goal $a$ in the world.

We now establish the completeness of the HTN-MAKER algorithm. We say that a set $M$ of HTN methods is ***complete*** relative to a set of goals $g$ if, for any classical planning problem $(s_0, g, O)$ that is solvable, the HTN-equivalent problem $(s_0, T, O, M)$ is solvable.

**Theorem 2.** *Let $O$ be a set of planning operators and $\tau$ be a set of annotated tasks for a classical planning domain.*

*Then, there exists a finite number of input classical planning problems and their solutions such that HTN-MAKER learns from this input a set $M$ of methods that is complete relative to $g$.*

A sketch of the proof follows. First, we note that the methods learned by HTN-MAKER from a given classical planning problem can be used to solve the equivalent HTN planning problem, because the methods learned may be decomposed in reverse order to produce the input plan, which is already known to be a solution. Second, we note that adding additional methods to an HTN domain cannot decrease the number of problems that can be solved with it. This follows from the HTN planning process, which at each choice point non-deterministically chooses an applicable method for decomposition. Adding additional methods may increase the number of choices, but any choice that had previously been available will remain so. Finally, the number of constant symbols in a classical planning domain is finite. Thus there are a finite number of possible atoms and a finite number of states. Thus, there are a finite number of classical planning problems in a given planning domain relative to a set of goals. Therefore, Theorem 2 holds.

Note that although the worst case requires learning from every problem in the domain, our experiments indicate that far fewer problems are needed in practice. In one experiment in the Logistics domain, we were able to solve all solvable problems that require delivering a single package to a location after learning from only six controlled pairs of problems and solution plans.

HTNs provide a planning language that is strictly more expressive than classical planning (Erol, Hendler, and Nau 1996): in the general case this language is context-free, whereas classical planning is restricted to regular languages. Since learning context-free HTNs is a very hard problem, we focus on a class of planning problems that are strictly more expressive than classical problems, yet less expressive than general HTNs. In the rest of this section, we formalize this class of planning problems and show that HTN-MAKER's learned HTNs induce planning problems in this class, although the algorithm learns HTN methods from classical planning problems and their solutions. Consider again the Logistics domain from the previous section. Suppose we have a planning problem in which a package is at an initial location $L0$, and needs to be delivered first to the location $L1$, then to $L2$, and finally, to $L3$. This planning problem cannot be represented as a classical planning problem without introducing function symbols into the representation language or fixing in advance the maximum number of intermediate locations for all problems. On the other hand, it would easily be represented as an HTN planning problem, where we have a task in the initial task list of the HTN planning problem for delivering the package to each of its goal locations in the specified order.

We formalize the notion of the class of planning problems such as the one above as follows. Let $(s_0, g, O)$ be a planning problem. Then, there exists a partition $(g_0, g_1, \ldots, g_k)$ of the conditions in $g$ such that each planning problem in the sequence $(s_0, g_0, O), (s_1, g_1, O), \ldots, (s_k, g_k, O)$ is a classical planning problem and the condition $g_i$ holds in the state $s_{i+1}$ for $i = 0, \ldots, k-1$.

We call the class of planning problems that have multiple goals but no classical representation as ***classically-partionable*** planning problems. Note that classically-partionable planning problems appear in many planning domains, including Logistics, Blocks-World, ZenoTravel, Rovers, Storage, and others that were used as benchmarks in the past International Planning Competitions. Many learning systems, such as Icarus (Langley and Choi 2006), cannot learn HTN methods for classically-partionable planning problems since Icarus learns HTN methods to "achieve a classical goal" and it needs the classical goal statement in its input due to its means-ends analysis. One way to enable Icarus to learn HTN methods for a classically-partionable planning problem is to re-factor the problem into a series of classical problems and give each problem as input to Icarus. However, this would require a supervisor system that would do the translation, run Icarus on the subproblems, and combine the results. HTN-MAKER, on the other hand, can learn from an initial state and a sequence of actions applicable in that state, without requiring a goal as input.

The following theorem establishes our expressivity result:

**Theorem 3.** *Let $O$ be a set of planning operators and $\tau$ be a set of annotated tasks for a classical planning domain. Suppose HTN-MAKER is given a set of classical planning problems in the domain and a solution plan for each of those planning problems, and it produces a set $M$ of HTN methods that are complete relative to $\tau$.*

*Let $(s, g, O)$ be a classically-partionable planning problem with the partition $(s_0, g_0, O), (s_1, g_1, O), \ldots, (s_k, g_k, O)$. Then, the HTN planning problem $(s, T, O, M)$ is solvable, where $T$ is a sequence of tasks $t_1, \ldots, t_k$ and there is an annotated task for each $i = 1, \ldots k$ in the domain.*

## Implementation and Experimental Evaluation

There are a number of implementation details that control the effectiveness of HTN-MAKER. First, our implementation of the choice of a method instantiation to add as a subtask when several are available selects the instantiation that extends over the largest portion of the underlying plan. This favors learning right-recursive methods. Secondly, our implementation of HTN-MAKER completely avoids learning left-recursive methods by requiring the first subtask to always be primitive. This is useful because an HTN planner like SHOP is likely to infinitely recurse on a left-recursive learned method if the first compound subtask of the method matches the head of the method. Finally, HTN-MAKER never learns a method from a segment of a plan if knowledge already exists to accomplish the task from there or earlier.

We first used the Logistics domain from the 2nd International Planning Competition (IPC-2) for our experiments. In Logistics, the objective is to deliver packages between locations in various cities using trucks and airplanes. We have varied the number of packages between 1 and 4 and randomly generated 100 planning problems. For all of these problems, we have used a single annotated task that delivers a package to a location. We have run 5 trials in which 75 of these problems are randomly selected as the training set and 25 are held out for testing. HTN-MAKER begins with an empty set of methods and updates it with methods learned from plans for each of the training problems presented in a random order. After learning from a training problem, we attempted to solve the HTN equivalent of each of the 25 test problems in SHOP using the methods learned thusfar.

Figure 4 shows the percentage of test problems solved by SHOP using the learned methods after each successive training example, averaged over the 5 trials. After learning from the first 25 training problems, the learned set of HTN methods rapidly converged to nearly full coverage — i.e., SHOP was able to solve most of the test problems with the methods learned by HTN-MAKER. After 60 training examples all test problems can be solved.

In order to test HTN-MAKER in a domain where the solution plans do not have the above characteristics, we have performed the following experiments with the Blocks-World domain, also from IPC-2. In Blocks-World, the solution plans are usually specific to particular configurations of the blocks in the initial state of a planning problem; thus, they allow for only limited generalizations into learned HTN methods.

To be able to perform this test in a controlled way, we have used a goal configuration that involves stacking $5$ blocks on top of each other. Then, we have generated all of the $561$ possible planning problems with $5$ blocks and this particular goal configuration. As before, we held out $25\%$ of these as test cases and used the same experimental design. In our experiments, HTN-MAKER learned much more slowly in the Blocks-World domain. In fact, using knowledge learned from the entire training set, SHOP was still unable to solve $40\%$ of the test set.

We also performed a similar experiment with $100$ problems from the Satellite domain from IPC-3. The majority of this domain could be learned from a single example, but there were some trials in which not enough knowledge was learned to solve the entire test set.

The Logistics and Satellite domains have a fairly straightforward structure; every goal can be solved by a plan that, other than different variable bindings, is quite similar to a plan that solves a different goal. Thus, a small set of training examples provides a pattern that can be adapted to solve most potential problems in the domain. The Blocks-World domain does not have this predictable structure because its goals are much more inter-dependent. Thus, it is quite difficult to generalize a solution to one problem in such a way that it can be used to solve a different problem. There are, however, a number of special cases that must be individually observed. These differences are demonstrated both in the convergence rates shown in Figure 4, and in the average number of methods learned during a trial, which is $40.4$ for Logistics, $20.0$ for Satellite, and $449.8$ for Blocks-World.

Even in the highly structured domains there are rare special cases that must be observed in the training set before they can be solved. This explains why our experiments in the Satellite domain do not show full convergence.

We have also experimentally verified that if we train on all $561$ Blocks-World problems described above, it is possible to solve all of them with the resulting knowledge. This confirms Theorem 2, our completeness result, and demonstrates the worst-case that we discussed above. Furthermore, in the Logistics domain we were able to learn a set of HTN methods that are complete relative to the goal of delivering a single package from 6 carefully selected problems and solutions.

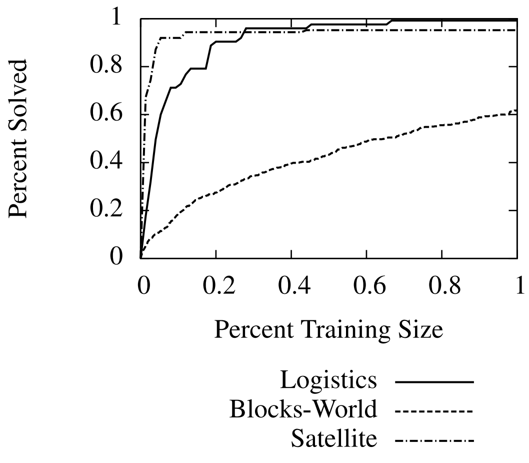

Figure 4: Percentage of problems solvable through the learning process.

## Related Work

Learning task decompositions means eliciting the hierarchical structure relating tasks and subtasks. Existing work on learning hierarchies elicits a hierarchy from a collection of plans and from a given action model (Nejati, Langley, and Könik 2006; Reddy and Tadepalli 1997; Ruby and Kibler 1991). A particularity of the existing work on learning task models is that the tasks from the learned hierarchies are the same goals that have been achieved by the plans. Reddy and Tadepalli's 1997 X-Learn, for example, uses inductive generalization to learn task decomposition constructs, which relate goals, subgoals, and conditions for applying d-rules. By grouping goals in this way, the learned task models led to speed-up in problem-solving. In (Nejati, Langley, and Könik 2006; Reddy and Tadepalli 1997), some of the problems that were solved using the learned task models could not be solved without using them (e.g., by using only the action models), but this was only because of improved performance with limited resources.

Two studies (Ilghami et al. 2005; Xu and Muñoz-Avila 2005) propose eager and lazy learning algorithms, respectively, to learn the preconditions of HTN methods. These systems require as input the hierarchical relationships between tasks and learn only the conditions under which a method may be used. Another recent work (Nejati, Langley, and Könik 2006) learns a special case of HTNs known as teleoreactive logic programs. Rather than a task list, this system uses a collection of Horn clause-like concepts. The means-end reasoning that is tightly integrated with this learning mechanism is incapable of solving problems that general HTNs can solve, such as the register assignment problem.

Work on learning macro-operators (e.g., (Mooney 1988; Botea, Muller, and Schaeffer 2005)) falls in the category of speed-up learning, as do work on learning search control knowledge ((e.g., (Minton 1998; Fern, Yoon, and Givan 2004)). Search control knowledge does not increase the number of problems that theoretically can be solved. However, from a practical stand point, these systems increase the number of problems that can be solved because of the reduction in runtime. Other researchers assumed that hierarchies are given as inputs for learning task models.

Inductive approaches have been proposed for learning from action models. For example, the DISTILL system learns domain-specific planners from an input of plans that have certain annotations (Winner and Veloso 2003). The input includes the initial state and an action model. DISTILL elicits a programming construct for plan generation that combines the action model and search control strategies. However, the strategies learned are not in the form of HTNs.

Another related work is abstraction in planning such as the Alpine (Knoblock 1993) and the Paris (Bergmann and Wilke 1995) systems. These systems use both a collection of operators and an abstraction model that indicates how to abstract and specialize plans. The systems are able to solve problems by abstracting them, finding abstract plans for the abstract problems, and specializing the abstract plans to generate concrete solution plans. It is conceivable that some of the ideas used in HTN-MAKER could be adapted to the problem of learning abstraction models.

## Conclusions

HTN planning is an effective problem-solving paradigm, but the high knowledge engineering cost of developing an HTN domain description is a significant impediment to the adoption of HTN planning technology. We have described a new algorithm, HTN-MAKER, for incrementally learning HTN domain knowledge from initial states of classical planning problems and solution plans for those problems. HTN-MAKER produces a set of HTN methods by learning the decomposition structure of tasks from annotated tasks and plans. The learner constructs a hierarchy in a bottom-up manner by analyzing the sequences of operators in a plan trace and determines the preconditions of methods from those of their subtasks.

We have presented theoretical results showing that the methods learned by HTN-MAKER are sound and complete relative to the set of goals for which annotated tasks are provided. Furthermore, these methods can be used to solve problems that could not be expressed using the classical planning knowledge from which they were learned. Our experiments on three well-known planning domains, Logistics, Blocks-World, and Satellite, demonstrated that HTN-MAKER converged to a set of HTN methods that solve nearly all problems in the domain as more problems are presented.

We intend to expand this work in a couple of future directions. First is to study the effects of providing additional information in the annotated tasks on convergence speed. Secondly, we are currently developing techniques for using reinforcement learning mechanisms on top of HTN-MAKER in order to learn the expected values of the HTN methods produced by the algorithm. This will enable us to study optimality and usefulness properties of the learned HTNs.

## Acknowledgments

This research was in part supported by the National Science Foundation (NSF 0642882) and the Defense Advanced Research Projects Agency (DARPA). The opinions in this paper are those of the authors and do not necessarily reflect the opinions of the funders.

## References

Bergmann, R., and Wilke, W. 1995. Building and refining abstract planning cases by change of representation language. *Journal of Artificial Intelligence Research* 53–118.

Botea, A.; Muller, M.; and Schaeffer, J. 2005. Learning partial-order macros from solutions. In *Proc. of ICAPS '05*. AAAI Press.

Erol, K.; Hendler, J.; and Nau, D. S. 1996. Complexity results for hierarchical task-network planning. *Annals of Mathematics and Artificial Intelligence* 18:69–93.

Fern, A.; Yoon, S. W.; and Givan, R. 2004. Learning domain-specific control knowledge from random walks. In *Proc. of ICAPS '04*. AAAI Press.

Ghallab, M.; Nau, D.; and Traverso, P. 2004. *Automated Planning: Theory and Practice*. Morgan Kauffmann.

Ilghami, O.; Muñoz-Avila, H.; Nau, D.; and Aha, D. W. 2005. Learning approximate preconditions for methods in hierarchical plans. In *Proc. of ICML '05*.

Knoblock, C. 1993. *Abstraction Hierarchies: An Automated Approach to Reducing Search in Planning*. Norwell, MA: Kluwer Academic Publishers.

Langley, P., and Choi, D. 2006. Learning recursive control programs from problem solving. *J. Mach. Learn. Res.* 7:493–518.

Minton, S. 1998. *Learning Effective Search Control Knowledge: an Explanation-Based Approach*. Ph.D. Dissertation, Carnegie Mellon University.

Mitchell, T.; Keller, R.; and Kedar-Cabelli, S. 1986. Explanation-based generalization: A unifying view. *Machine Learning* 1.

Mooney, R. J. 1988. Generalizing the order of operators in macro-operators. *Machine Learning* 270–283.

Nau, D.; Cao, Y.; Lotem, A.; and Muñoz-Avila, H. 1999. SHOP: Simple hierarchical ordered planner. In *Proc. of IJCAI '99*, 968–973. AAAI Press.

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

Nejati, N.; Langley, P.; and Könik, T. 2006. Learning hierarchical task networks by observation. In *Proc. of ICML '06*, 665–672. New York, NY, USA: ACM.

Reddy, C., and Tadepalli, P. 1997. Learning goal-decomposition rules using exercises. In *Proc. of ICML '97*.

Ruby, D., and Kibler, D. F. 1991. SteppingStone: An empirical and analytic evaluation. In *Proc. of AAAI '91*, 527–531. Morgan Kaufmann.

Sacerdoti, E. D. 1975. The nonlinear nature of plans. In *Proc. of IJCAI '75*, 206–214.

Winner, E., and Veloso, M. M. 2003. DISTILL: Learning domain-specific planners by example. In *Proc. of ICML '03*.

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