Learning Hierarchical Task Networks for Nondeterministic Planning Domains

Chad Hogg[1] and **Ugur Kuter**[2] and **Héctor Muñoz-Avila**[1]

[1]Department of Computer Science & Engineering, Lehigh University,
Bethlehem, Pennsylvania 18015, USA

[2]Institute for Advanced Computer Studies, University of Maryland,
College Park, Maryland 20742, USA

Abstract

This paper describes how to learn Hierarchical Task Networks (HTNs) in nondeterministic planning domains, where actions may have multiple possible outcomes. We discuss several desired properties that guarantee that the resulting HTNs will correctly handle the nondeterminism in the domain. We developed a new learning algorithm, called HTN-MAKERND , that exploits these properties. We implemented HTN-MAKERND in the recently-proposed HTN-MAKER system, a goal-regression based HTN learning approach. In our theoretical study, we show that HTN-MAKERND soundly produces HTN planning knowledge in low-order polynomial times, despite the nondeterminism. In our experiments with two nondeterministic planning domains, ND-SHOP2, a well-known HTN planning algorithm for nondeterministic domains, significantly outperformed (in some cases, by about 3 orders of magnitude) the well-known planner MBP using the learned HTNs.

1 Introduction

The problem of planning in nondeterministic domains, where one or more actions may have multiple possible outcomes, is of recurring and increasing interest among researchers. For example, a node in a computer network may attempt to pass a message to another node, but this message may or may not arrive due to connectivity issues. Despite recent advances [Jensen *et al.*, 2001; Cimatti *et al.*, 2003; Pistore and Traverso, 2001], nondeterministic planning problems, are still very hard to solve in practice.

Some researchers have proposed to use domain knowledge encoded as HTNs (or similar representations) for nondeterministic domains [Karlsson, 2001; Kuter and Nau, 2004; Kuter *et al.*, 2008]. It has been demonstrated that this approach achieves a significant reduction in the search spaces of a planner that can exploit such knowledge compared to a planner that does not. However, it is usually difficult to compile expert-level HTN knowledge that leads to superior performance in nondeterministic planning domains; the HTNs must specify which actions to take in a potentially exponential number of circumstances in order to generate a solution policy for a nondeterministic planning problem. Although in some toy domains this is feasible, in many cases it is not.

Recent advances in *deterministic* planning domains have shown that HTN-based planning knowledge can be effectively learned in an automated fashion and used in planners to generate solution plans efficiently. Examples of such HTN-learning systems include [Langley and Choi, 2006; Nejati *et al.*, 2006; Hogg *et al.*, 2008]. However, research for learning planning knowledge in *nondeterministic* domains thusfar has only concentrated on action models [Pasula *et al.*, 2004; Jensen and Veloso, 2007], and no HTN-learning algorithms have been developed for such planning domains.

This paper focuses on learning HTNs in nondeterministic domains. We found that if HTNs have certain properties, then they will be able to reason about the nondeterministic effects of the actions; otherwise, the planning knowledge may not be useful to generate solutions for nondeterministic planning problems. These properties describe (1) a particular kind of recursive structure that is guaranteed to generate a plan for every nondetermistic outcome of an action, and (2) a way to compute the applicability conditions for those structures.

We describe how to learn HTNs that commit to the above properties. We implemented our ideas in the recently-proposed HTN-MAKER learning system [Hogg *et al.*, 2008], which uses goal-regression mechanisms to learn HTNs originally in *deterministic* domains. We present the outcome of this work, a learning system that we call HTN-MAKERND .

In our theoretical study, we prove the soundness and completeness of HTN-MAKERND . We also show that HTN-MAKERND generates HTN knowledge in low-order polynomial times with respect to the number of input plan traces and the maximum length of those traces. Our experimental evaluation in two well-known nondeterministic planning domains confirms the theoretical results. We compared the ND-SHOP2 algorithm [Kuter and Nau, 2004] using our learned HTNs with the well-known planning algorithm MBP [Cimatti *et al.*, 2003]. The experiments showed that the former was able to outperform the latter significantly (in some cases, by about 3 orders of magnitude) with the learned HTNs.

2 Preliminaries

Nondeterministic Planning Domains. Intuitively, a *nondeterministic planning domain* is one in which each action may have more than one possible outcome. Formally, it is a triple

$D = (\mathcal{S}, A, \gamma)$, where $\mathcal{S}$ is a finite set of *states*, A is a finite set of *actions*, and $\gamma : \mathcal{S} \times A \rightarrow 2^{\mathcal{S}}$ is the *state-transition function*. An action a is *applicable* in s if $\gamma(s, a)$ is nonempty.

A *policy* π is a function from a set $S_\pi \subseteq \mathcal{S}$ into A, such that for each $s \in S_\pi$, the action $\pi(s)$ is applicable to s. The *execution structure* of π is a digraph Σ_π representing all possible executions of π. Formally, $\Sigma_\pi = (V_\pi, E_\pi)$, where $V_\pi = S_\pi \cup \bigcup\{\gamma(s, \pi(s)) \mid s \in S_\pi\}$ and $E_\pi = \{(s, s') \mid s \in S_\pi, s' \in \gamma(s, \pi(s))\}$. That is, the vertices of the execution structure are the states for which an action is defined and any final states that can be reached by following such an action, and there is an edge from each state for which an action is defined to each state that can be reached by applying that action to that state. If there is a path in Σ_π from s to s' then s is a π-*ancestor* of s' and s' is a π-*descendant* of s. If the path has length 1, s is a π-*parent* of s' and s' is a π-*child* of s. A node is a *leaf* if it has no π-children.

A nondeterministic planning problem is a triple $P = (D, S_0, G)$, where $D = (\mathcal{S}, A, \gamma)$ is a nondeterministic planning domain, $S_0 \subseteq \mathcal{S}$ is a set of initial states, and $G \subseteq \mathcal{S}$ is a set of *goal states*. P may have several kinds of solutions as follows [Cimatti *et al.*, 2003]. A *weak solution* for P is a policy π such that every state $s \in S_0$ is a π-ancestor of at least one goal state. A *strong-cyclic solution* is a policy π such that every state s in Σ_π is a π-ancestor of at least one goal state and the leaf nodes in Σ_π are goal states. Note that a strong-cyclic solution is allowed, but not required, to contain cycles. A *strong solution* is a policy π such that π is acyclic, every state s in Σ_π is a π-ancestor of at least one goal state, and the leaf nodes in Σ_π are goal states.

Hierarchical Task Networks (HTNs). We use the usual definitions for HTNs and HTN planning as in Chapter 11 of [Ghallab *et al.*, 2004]. A ***task*** is a symbolic representation of an activity in the world. We formalize a task 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 [Hogg *et al.*, 2008] an ***annotated task*** was defined as a triple $(\mathsf{Task}, \mathsf{Pre}, \mathsf{Eff})$ where Task is a task, Pre are some conditions that hold in any state from which the task can be accomplished, and Eff are the positive effects that will be true in any state where the task has been accomplished. We generalize these preconditions and effects as a triple (t, S_i, S_f) where t is a task symbol, $S_i \subseteq \mathcal{S}$ is a set of states from which the task may be accomplished, and $S_f \subseteq \mathcal{S}$ is a set of states in which the task has been accomplished. Given a set $G \subseteq \mathcal{S}$ of states in which some goals are true, we define the ***equivalent annotated task*** to those goals as one in which $(t, \mathcal{S}, G)$ for some task symbol t. That is, the annotated task equivalent to some set of goals can be attempted from any state and is completed when it reaches any state in which those goals hold. The ***equivalent HTN problem*** to a nondeterministic planning problem P has the same initial state as P and a single initial task that is the equivalent annotated task to the goals of P. An ***HTN method*** is a description of how to decompose nonprimitive tasks into simpler ones. Formally, a ***method*** is a partial function $m : \mathcal{S} \times \mathcal{T} \rightarrow \mathcal{H}$, where $\mathcal{T}$ is a finite set of tasks and $\mathcal{H}$ is a set of sequences of tasks. Given a state s and a task t, if $m(s, t)$ is defined then we say that m is applicable in s to the task t. The result of applying m in s to t is the sequence of the subtasks specified by $m(s, t)$.

3 Learning HTNs for Nondeterministic Domains

Existing well-known algorithms for learning HTN methods in *deterministic* domains, such as X-Learn [Reddy and Tadepalli, 1997], Icarus [Nejati *et al.*, 2006], and HTN-MAKER [Hogg *et al.*, 2008], have two mechanisms in common:

Bottom-up Learning. Learning proceeds in a bottom-up fashion by grouping a sequence of actions and previously learned structures $a_1 \ldots a_n$ in an input trace together as subtasks of newly constructed methods.

Precondition Identification. Applicability conditions for an HTN method are computed. For example, X-learn uses explanation-based learning techniques to prune conditions not needed in the state S preceeding the first action a_1 in the trace of actions $a_1 \ldots a_n$. Similarly, HTN-MAKER uses the concept of goal regression from explanation-based learning to collect all necessary preconditions in the trace that would allow execution of the actions $a_1 \ldots a_n$.

We now illustrate how these mechanisms are used to learn HTNs in the deterministic case, where they fail in the nondeterministic case, and how to extend these mechanisms to correctly learn HTNs in that case. We then exemplify these extended mechanims with the HTN-MAKER algorithm and call the resulting HTN learner HTN-MAKER$^{\text{ND}}$.

3.1 Learning HTN Knowledge in Deterministic Domains: An Illustration

Before describing the extensions, we discuss an example in the traditional 4-operator version of the Blocks World domain. Here, blocks sit on a table and may be picked up from the table, put down onto the table, stacked on top of another block, or unstacked from above another block by a gripper.

Consider a simple input trace, which we will call **simple stack**, consisting of the following sequence of deterministic actions: (PICKUP a) (STACK a b). Assume that originally block a was clear and on the table, block b was clear, and the gripper was empty. Then after executing these actions block a will be on top of b and the gripper will be empty once again. An HTN learning algorithm will learn the above sequence to achieve the task of stacking a on top of b. It will also learn as applicability conditions the same conditions above. The two actions will correctly solve any deterministic Blocks World problem requiring to stack a block x on top of a block y, assuming that x and y meet the above conditions.

3.2 Precondition Identification and Bottom-Up Learning, Revisited for Nondeterminism

Consider the nondeterministic version of Blocks World as described in [Kuter and Nau, 2004]. In this version, an action may unexpectedly drop the block on the table. That is, each of the operators (STACK $?x$ $?y$), (UNSTACK $?x$ $?y$), and (PICKUP $?x$) have an additional possible effect that specifies the case in which the block $?x$ is dropped on the table. Additionally, each of the four operators may have no effect at all.

In the nondeterministic version of Blocks World, the **simple stack** trace is a valid trace (one in which each action has its usual effects). If this trace is given as input to an HTN learning algorithm in a nondeterministic version of the domain, it would learn the same methods as before. However, these methods would fail to generate a strong or strong-cyclic solution because the policy does not specify any action if (PICKUP a), (STACK a b), or both fails.

Now assume that the following two additional traces are given, which succesfully stack the block a on top of b from the same initial state as the simple stack trace:

- **Failed pickup**. (PICKUP a) (PICKUP a) (STACK a b), in which the first pickup action fails and needs to be repeated.
- **Failed stack**. (PICKUP a) (STACK a b) (PICKUP a) (STACK a b), in which the first stack action fails and the block a needs to be picked up again and stacked.

The two traces above together with the simple stack trace provide the information needed by an HTN learner to generate methods to correctly handle the different outcomes of the pickup and stack actions. A learning algorithm that fulfills the following two requirements will ensure that these outcomes are learned from the three traces.

Right-recursive Bottom-Up Learning of Task Pairs. The methods learned consist of one or two subtasks. If there is one subtask, then it must be primitive. If there are two subtasks, then the first must be primitive and the second must be nonprimitive. The selection of a method to decompose the second subtask will handle the fact that the first subtask has many possible outcomes, and will allow the planner to select an appropriate method for each.

Suppose a sequence of actions $a_1 \ldots a_n$ in an input trace p together achieves a task t. Then, we would require the following n methods, each having t as its head task:

- The method m_n has a single subtask a_n (i.e., the terminal case).
- For i between 1 and $n - 1$, each method m_i has two subtasks: a_i and t (i.e., the recursive case).

Step-wise Precondition Identification. Applicability conditions must be collected for each method m_i in a stepwise fashion, taking into account which of the nondeterministic outcomes was triggered for each action a_i in the trace.

- The conditions for m_n are collected by determining the applicability conditions for a_n and m_{n+1} relative to trace p.
- For i between 1 and $n - 1$, the preconditions for each method m_i are obtained by determining the applicability conditions for a_i relative to trace p.

These extensions enable a learning algorithm to incrementally learn the various possible outcomes of action a_i. In the nondeterministic Blocks World example, the simple stack trace will learn two methods for a task t. In Icarus t would be simply the goal to achieve, *(on ?a ?b)*. Equivalently, in HTN-MAKER it would be an annotated task, say *(Make-2Pile ?a ?b)*, having no preconditions and the single effect *(on ?a ?b)*. The methods would have the form: *m1 = (task: t preconditons: p1 subtasks: (PICKUP ?a), t)* and *m2 = (task: t preconditons: p2 subtasks: (STACK ?a ?b))*. Interestingly, these methods already cover the PICKUP failure indicated in the failed pickup trace because if (PICKUP a) fails, the state of the world will revert to the situation where method $m1$ is applicable and therefore it can pickup block a again. For the same reasons it covers multiple subsequent failures of the pickup action in this state. Both Icarus and HTN-MAKER provide mechanisms to avoid learning new methods in this situation. For example, HTN-MAKER can check whether new methods are subsumed by existing methods by checking if their tasks and subtasks match and if the preconditions of one are implied by those of the other.

The failed stack trace provides new information, namely what to do if the stack operation fails. It will learn the method: *m3 = (task: t preconditons: p3 subtasks: (STACK ?a ?b), t)*, which is learned from the second action of the failed stack trace. All other methods that could be learned from the failed stack trace are already subsumed by $m1$ and $m2$ and therefore aside from $m3$ no new methods would be learned from this trace. These 3 methods would allow ND-SHOP2 to generate a strong solution policy for any stacking task (STACK $?a$ $?b$) provided that $?a$ is originally on the table, $?a$ is clear, $?b$ is clear, and the gripper is empty.

We believe that any of the known HTN learners could produce methods of this form with appropriate input or modifications to restrict what may be learned. For illustration purposes we have chosen to implement our ideas in HTN-MAKER .

Algorithm 1 HTN-MAKER$^{\text{ND}}$ (p, S_0, τ, M_0)

```
1:  $M \leftarrow M_0$
2:  for $S' \leftarrow S_0$ to state after last action in $p$ do
3:      for $S'' \leftarrow S'$ to $S_0$ do
4:          Select a task $t \in \tau$ such that
            - the effects $e_t$ of $t$ are satisfied in $S'$ and
            - the preconditions of $t$ are satisfied in $S''$
5:          $(a_1 \ldots a_n) \leftarrow$ collectActions$(S'', S', p)$
6:          $c \leftarrow$ regressConditions$(a_n, e_t)$
7:          for $k \leftarrow n$ to 1 do
8:              Add to $M$ if not already there a new method:
                  if $k = n$ then
                    task: $t$, preconditions: $c$, subtasks: $a_n$
                  else
                    task: $t$, preconditions: $c$, subtasks: $a_k, t$
9:              $p \leftarrow$ regressConditions$(a_{k-1}, c)$
10: Return $M$
```

3.3 Nondeterministic Learner

Algorithm 1 shows a high-level description of the top-level learning procedure of HTN-MAKER$^{\text{ND}}$ that incorporates the two extensions discussed before. The inputs are an input trace

p, the initial state from which that trace may be executed S_0, a set of annotated tasks τ, and a (possibly empty) set of methods M_0. It returns a collection of methods M that includes those given to it and any new ones learned from the trace.

In Algorithm 1, S' indicates the state in which the task t has been accomplished, while S'' is the state from which the accomplishment of that task began. The actions that accomplished t are $(a_0 \ldots a_n)$, while the preconditions for the action sequence $(a_k \ldots a_n)$ are c. For each of the actions in $(a_0 \ldots a_n)$, a method is learned that satisfies the above requirements.

A set of traces and initial states may be processed by a sequence of instantiations of HTN-MAKER$^{\mathrm{ND}}$ such that the set of methods produced in one iteration becomes part of the input to the next iteration.

4 Properties

Let D be a nondeterministic planning domain and τ be a set of annotated tasks for D. Let $P = (D, S_0, G)$ be a nondeterministic planning problem in D and π be a solution for P.

Theorem 1 *Suppose HTN-MAKERND is given a set of execution traces for D and it produces a set M of HTN methods. Let $\mathcal{A}$ be a sound nondeterministic HTN planning algorithm. Given the HTN-equivalent problem of P with the methods M, suppose $\mathcal{A}$ generates a weak, strong, or strong-cyclic policy π. Then, π is a weak, strong, or strong-cyclic solution for P.*

Sketch of proof. For the strong-planning case, by our definitions, the input execution traces will include strong paths that do not include any cyclic execution. Suppose π is not a strong solution for P. There are two cases:

First, suppose that there is a cyclic path in π. Given the learned methods in M, the first case can happen only if the planning algorithm $\mathcal{A}$ uses methods learned for different purposes from different input execution traces. The only way a cycle could happen is when the learned methods induce a path in which there is a state s that is a π-ancestor of itself. However, since HTN-MAKER$^{\mathrm{ND}}$ is learning from acyclic paths, there must be a subset of methods in M that will induce an acyclic path. Thus, the planner $\mathcal{A}$ will eventually select those combinations of the methods and generate π. Otherwise, if $\mathcal{A}$ is sound, it would not generate π at all.

Second, suppose that π contains a state that is not a π-ancestor of a goal state. This situation above might happen if (i) there are infinite cycles in π or (ii) there is a terminal state in π that is not a goal state. The case (i) cannot happen for the reasons stated above. The case (ii) cannot happen because otherwise $\mathcal{A}$ would not generate π.

Therefore, the theorem follows for strong planning. The proofs for weak and strong-cyclic planning are very similar, but we omit them due to space limitations. $\square$

We now establish the completeness of the HTN-MAKER$^{\mathrm{ND}}$ algorithm. We say that a set M of HTN methods is ***complete*** relative to a set G of goals if when any nondeterministic planning problem (D, S_0, G) has a weak, strong, or strong-cyclic solution, the HTN-equivalent problem has also a weak, strong, or strong-cyclic solution.

Theorem 2 *There exists a finite number of input execution traces such that HTN-MAKERND learns from this input a set M of methods that is complete relative to G.*

This is shown to be true in the deterministic case in [Hogg *et al.*, 2008]. The nondeterministic case is similar; there are an infinite number of possible traces in a domain, but HTN-MAKER$^{\mathrm{ND}}$ needs only to see one example of each cycle that makes this the case.

Theorem 3 *Let $\mathcal{P}$ be the set of execution traces for π. Then, given $\mathcal{P}$, HTN-MAKERND will return a set of methods M in time $O(|\mathcal{P}|n^3)$, where n is the number of actions in the longest execution trace in $\mathcal{P}$ and $|\mathcal{P}|$ denotes the number of execution traces in $\mathcal{P}$.* [1]

Sketch of proof. Given an execution trace p of length n, HTN-MAKER$^{\mathrm{ND}}$ searches every possible prefix of the trace (see Line 2 of Algorithm 1) of length $1, \ldots, n$. Within each plan prefix, HTN-MAKER$^{\mathrm{ND}}$ then generates every possible postfix of that plan prefix (see Line 3). Finding a postfix is therefore an $O(n^2)$ operation, where n is the number of actions in the execution trace. For each postfix, HTN-MAKER$^{\mathrm{ND}}$ learns a set of methods by first finding the actions in that postfix via the collectActions subroutine (see Line 5), which has the time complexity of $O(n)$, and then regressing the predicates that appear at the final state of the postfix toward the start state of it using the preconditions and effects of the actions (see Line 6). This procedure has a time complexity of $O(1)$ since it simply stores regressed condition given the goal in its second parameter through the action in its first parameter. Finally, HTN-MAKER$^{\mathrm{ND}}$ generates n methods (see Lines 7–9) from the processed postfix. These n methods implement the generation of right-recursive methods that are required for nondeterministic domains, as we discussed above. Each of these methods has either a single primitive subtask or a primitive subtask followed by a nonprimitive subtask. The generation process has a complexity of $O(n)$. Thus, the entire HTN-MAKER$^{\mathrm{ND}}$ procedure has a complexity of $O(n^3)$. For a set of input $|\mathcal{P}|$ traces, the theorem follows. $\square$

5 Implementation and Experimental Evaluation

As mentioned above, we have implemented the HTN-MAKER$^{\mathrm{ND}}$ algorithm in the HTN-MAKER system [Hogg *et al.*, 2008]. We conducted an experimental evaluation of HTN-MAKER$^{\mathrm{ND}}$ in two benchmark nondeterministic planning domains, namely the Robot Navigation domain [Cimatti *et al.*, 2003] and the nondeterministic Blocks World domain [Kuter and Nau, 2004], which we describe below.

We have used HTN-MAKER$^{\mathrm{ND}}$ to learn an HTN description of each of the nondeterministic planning domains. Then, we used that description in the ND-SHOP2 planner [Kuter and Nau, 2004], an HTN planner developed for nondeterministic domains. We compared the time performance of

[1] An option to HTN-MAKER$^{\mathrm{ND}}$ causes it to reject any method subsumed by one already known. When this option is used, the time complexity also depends on the number and size of methods known.

ND-SHOP2 using the learned HTNs with the MBP planner [Cimatti *et al.*, 2003], another state-of-the-art planning system that uses symbolic model-checking techniques for planning in nondeterministic domains. MBP does not learn from traces because it uses no knowledge beyond the input action descriptions. MBP encodes the action model, which is also given as input to our learner. MBP was used as a benchmark versus ND-SHOP2 in [Kuter and Nau, 2004]; thus, we also chose it to compare ND-SHOP2 with our learned HTNs. [2]

All experiments were performed on a Linux-based virtual machine with 512 MB of memory and a 2.16 GHz Intel Core Duo processor.

Robot Navigation. This domain is introduced as one of the benchmark domains for MBP [Cimatti *et al.*, 2003]. It consists of a building with 8 rooms connected by 7 doors, with several packages that need to be moved to desired locations and a robot that is responsible for moving them. The robot can open and close doors, move through doorways, and pick up and put down objects. It can hold at most one package at a time. Some or all of the doors are "kid doors," and a "kid" can open or close any combination of those doors after each of the robot's actions. The kid is modeled not as a separate agent, but as a set of nondeterministic effects for each action.

In this domain, we randomly generated 500 planning problems with 1 package to deliver. For each planning problem, we generated 50 random execution traces from the initial state of the problem to one of its goal state. We ran HTN-MAKER$^{\mathrm{ND}}$ on these 25,000 randomly-generated execution traces and produced an HTN domain description. We then generated 25 planning problems randomly, for each number n of packages, where $n = 1 \ldots 5$, as testing examples. We gave the learned HTN description to ND-SHOP2 and compared its performance with MBP on the test planning problems.

Figure 1 shows the results. MBP was only able to solve 3 of the 25 problems with 5 packages within a time limit of 1 CPU hour, so we are not reporting that datapoint in the Figure 1 above. ND-SHOP2 with the learned HTN domain description was able to solve almost all of the problems with $1 \ldots 5$ packages; however, our experiments showed that the learned HTNs did not cover one of the 1-package problems, two of the 2-,3-, and 4-package problems, and 6 of the 5-package problems. That is, the solutions to these problems required a combination of actions that was not observed in the training examples. The times for these problems are not included in the averages for MBP or ND-SHOP2. In the problems that the learned HTNs was not able to solve, ND-SHOP2's average running times before it returned failure were: 5.36 seconds for the 1-package case, 12.2 seconds for the 2-package case, 80.775 seconds for the 3-package case, 200.47 seconds for the 4-package case, and finally, 35.63 for the 5-package case.

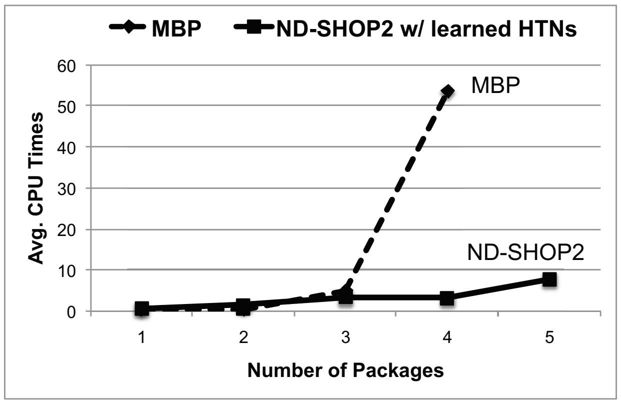

Figure 1: Average running times for ND-SHOP2 using the learned HTNs and MBP, as a function of increasing number of packages in the Robot Navigation domain.

Nondeterministic Blocks World. This is like the classical Blocks World, except that an action may have three possible outcomes: (1) the same outcome as in the classical case, (2) the block slips out of the gripper and drops on the table, and (3) the action fails completely and the state does not change. This domain was introduced in [Kuter and Nau, 2004] as a benchmark for ND-SHOP2.

In this domain, we randomly generated 1000 8-block problems, and for each problem, we generated 100 solution traces for a total of 100,000 training traces. We ran HTN-MAKER$^{\mathrm{ND}}$ on these traces and generated an HTN description of the domain. As the testing set, we randomly generated 50 problems for each size of problem (3 blocks through 8 blocks). As before, we ran ND-SHOP2 with the learned HTN domain and compared its performance against MBP on the test planning problems.

Figure 2 shows the results. Again, MBP was only able to solve 11 of the 50 problems with 8 blocks in the time alloted, so we are not reporting that datapoint in the Figure 2 above. ND-SHOP2 with the learned HTN domain description was able to solve all problems with 3, 4, and 5 blocks; the learned HTNs, however, did not cover 5 of the 6-block problems, 7 in the 7-block case, and 12 of the 8-block problems. The times for these problems are not included in the averages for MDP or ND-SHOP2. In the problems that the learned HTNs could not solve, ND-SHOP2's average running times before it returned failure were: 0.04 seconds for the 6- and 7-block cases, and 0.03 for the 8-block case.

Discussion. It is possible in both domains that there is some overlap between the training and testing sets, but this is very unlikely due to the very large number of potential problems from which each set was randomly selected. For example, there are more than 150 billion distinct 8-block problems in

[2]We used the above experimental setup in order to evaluate the learned HTNs by HTN-MAKER$^{\mathrm{ND}}$. In addition to the above, we've also investigated the possibility of a comparison between HTN-MAKER$^{\mathrm{ND}}$ and Icarus. However, Icarus does not generate strong or strong-cyclic solution policies in nondeterministic domains. Icarus would find an execution trace from the initial state to the goals and those plans could include cyclic executions (i.e., repeated action executions to achieve subgoals). However, strong or strong-cyclic policies require that from every state reachable from the initial state a goal must also be reachable. To the best of our knowledge, Icarus' skills do not necessarily guarantee such solutions, as the execution will stop when the goals are achieved. Thus, since a comparison between Icarus and HTN-MAKER$^{\mathrm{ND}}$ would not be fair to the latter, we have not performed such experiments.

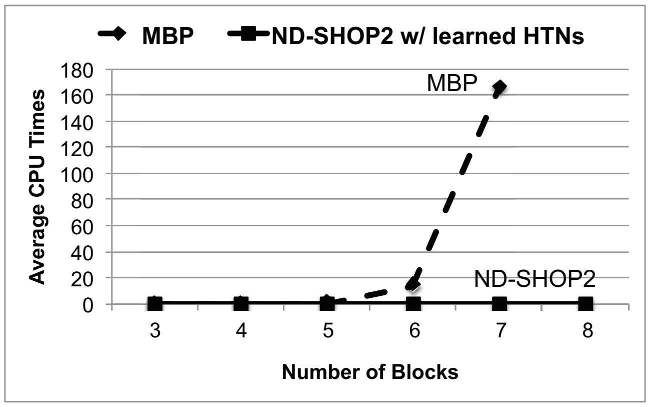

Figure 2: Average running times for ND-SHOP2 using the learned HTNs and MBP, as a function of increasing number of blocks in the nondeterministic Blocks World domain.

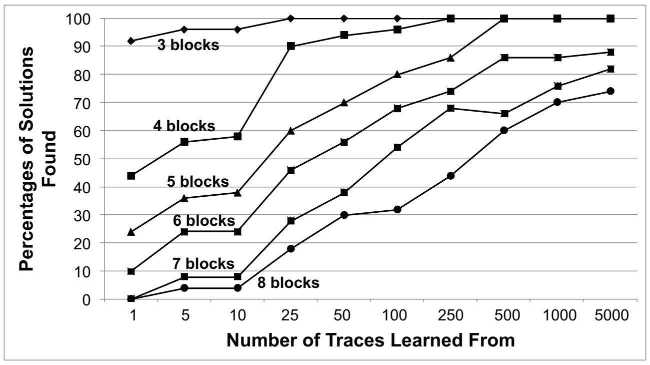

Figure 3: Learning rate for HTN-MAKERND in the nondeterministic Blocks World domain.

Blocks World.

Figure 3 shows the learning rate of HTN-MAKERND in the nondeterministic Blocks World domain. Each curve represents the percentage of the 50 testing problems of that size that were solvable after learning from a certain number of traces. There are few different problems containing only 3 or 4 blocks, so strategies to solve all problems may be learned from only a few examples. The number and complexity of problems increases exponentially with the number of blocks, so many more traces are required to cover every edge case. If our training set had included traces from more problems we would have eventually been capable of solving every problem in the testing set.

6 Related Work

Our work is closely related to Icarus [Langley and Choi, 2006], X-Learn [Reddy and Tadepalli, 1997], Stepping-Stone [Ruby and Kibler, 1991], and HTN-MAKER [Hogg *et al.*, 2008]. These systems aim to learn the hierarchical relations between tasks. Icarus, for example, groups actions into HTN methods based on a given set of concepts represented as Horn clauses. These concepts play a similar role to the input tasks in HTN-MAKERND, describing to the learner the target that it should aim to learn. However all of these works require deterministic actions, whereas in HTN-MAKERND actions might be nondeterministic.

Icarus can deal with reactive environments; i.e., the environment may change during execution as a result of external factors. Icarus's built-in learning and execution mechanism can learn from those situations by creating new HTN methods that fill the gaps in its current knowledge. This means that Icarus drops the static assumption of classical planning, namely, "No exogenous events: no changes except those performed by the controller". In contrast, HTN-MAKERND drops the deterministic assumption, namely, "each action or event has only one possible outcome" [Ghallab *et al.*, 2004]. [3] To the best of our knowledge, HTN-MAKERND is the first learner for nondeterministic planning domains.

Our work is also related to learning preconditions for HTNs such as [Ilghami *et al.*, 2005], [Xu and Muñoz-Avila, 2005], and [Yang *et al.*, 2007]. However, unlike HTN-MAKERND, those works assume that the HTNs are given and that the actions are deterministic. HTN-MAKERND learns both the HTNs and the applicability conditions simultaneously, although it is conceivable that some of these techniques could be used to simplify the preconditions learned by HTN-MAKERND. For example, [Ilghami *et al.*, 2005] describes how to use candidate elimination for finding the minimal sets of preconditions needed for applying a method.

In Reinforcement Learning [Sutton and Barto, 1998], the learning system aims to learn an optimal policy, maximizing some given utility, through a trial-and-error process in which actions are tried in the environment and the outcome of an action is evaluated to determine if selecting the action was useful or not in the current state. Reinforcement Learning systems can deal with nondeterministic actions (e.g., as in robotics tasks [Smart and Kaelbling, 2002]). It is conceivable that HTN-MAKERND could be extended to incorporate Reinforcement Learning techniques. This would require defining a utility function so that the utility of the intermediate states could be computed on the input traces. This would result in leaning methods that generate optimal policies rather than, as currently, learning methods that guarantee that the policies generated will reach desired states regardless of how well, relative to the utility, those states are reached.

7 Conclusions

We have studied the problem of learning HTN methods in nondeterministic domains in this paper. We identified principles of nondeterministic planning domains that are necessary for learning successful task decompositions that correctly account for multiple possible outcomes of the actions. Based on these principles, we have described two basic mechanisms: (1) learning methods that have either two subtasks, i.e., one primitive followed by a compound, or a single primitive task; (2) learning the preconditions of each method in (1) based on the outcome of each action that appears in the input trace.

[3]Reactive domains, under certain restictions, can be modeled as nondeterministic planning domains [Au *et al.*, 2008]. As an example in the Robot Navigation domain, the kid's actions are modeled as a nondeterministic effect of the robot's action for openning a door.

We have grounded these ideas in the HTN-MAKER learning system, which uses backwards-action chaining to generate the task-subtask relations in the methods and goal-regression to elicit the methods' preconditions. We call the algorithm we developed HTN-MAKERND. In our theoretical analysis of HTN-MAKERND, we showed that it is sound and complete. We also found that the algorithm runs in $O(|\mathcal{P}|n^3)$ time in the number $|\mathcal{P}|$ of input traces and their length n. We tested the HTN domain descriptions learned by HTN-MAKERND by using ND-SHOP2, an existing HTN planner for nondeterministic domains. In two standard benchmark domains, nondeterministic Blocks World and the Robot Navigation domain, using the learned HTNs, ND-SHOP2 was able to outperform another state of the art planner MBP by about 3 orders of magnitude. It should be noted that this speedup is not due to the learned HTNs being especially clever. Rather, our work allows the general advantages of HTNs to be gained in nondeterministic domains without requiring that a human encode appropriate methods.

In the future, we would like to study how to automatically learn methods that restrict the ordering of the subgoals to be achieved, as we believe this would yield significant reductions in the search space.

Acknowledgments

This work was supported in part by DARPA's Transfer Learning and Integrated Learning programs, AFOSR grants FA95500510298, FA95500610405, and FA95500610295, and the National Science Foundation (NSF 0642882). The opinions in this paper are those of the authors and do not necessarily reflect the opinions of the funders.

References

[Au *et al.*, 2008] T.-C. Au, U. Kuter, and D. Nau. Planning for interactions among autonomous agents. In *Proceedings of the International Workshop on Programming Multi-Agent Systems (ProMAS-08)*, 2008.

[Cimatti *et al.*, 2003] Alessandro Cimatti, Marco Pistore, Marco Roveri, and Paolo Traverso. Weak, strong, and strong cyclic planning via symbolic model checking. *Artificial Intelligence*, 147:35–84, 2003.

[Ghallab *et al.*, 2004] M. Ghallab, D. Nau, and P. Traverso. *Automated Planning: Theory and Practice*. Morgan Kauffmann, 2004.

[Hogg *et al.*, 2008] Chad Hogg, Héctor Muñoz-Avila, and Ugur Kuter. HTN-MAKER: Learning HTNs with minimal additional knowledge engineering required. In *AAAI-08*, 2008.

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

[Jensen and Veloso, 2007] R. M. Jensen and M. Veloso. Learning non-deterministic multi-agent planning domains. In *Proceedings of the ICAPS-07 Workshop on Planning and Learning*, 2007.

[Jensen *et al.*, 2001] R. Jensen, Manuela M. Veloso, and M. H. Bowling. OBDD-based optimistic and strong cyclic adversarial planning. In *Proceedings of the Sixth European Conference on Planning (ECP-01)*, 2001.

[Karlsson, 2001] Lars Karlsson. Conditional progressive planning under uncertainty. In *IJCAI-01*. Morgan Kaufmann, 2001.

[Kuter and Nau, 2004] Ugur Kuter and Dana Nau. Forward-chaining planning in nondeterministic domains. In *AAAI-04*, 2004.

[Kuter *et al.*, 2008] Ugur Kuter, Dana Nau, Marco Pistore, and Paolo Traverso. Task decomposition on abstract states for planning under nondeterminism. *Artificial Intelligence, Special Issue on Advances in Automated Planning*, 2008.

[Langley and Choi, 2006] Pat Langley and Dongkyu Choi. Learning recursive control programs from problem solving. *Journal of Machine Learning Research*, 7:493–518, 2006.

[Nejati *et al.*, 2006] N. Nejati, P. Langley, and T. Könik. Learning hierarchical task networks by observation. In *ICML-06*, 2006.

[Pasula *et al.*, 2004] H. M. Pasula, L. S. Zettemoyer, and L. P. Kaebling. Learning probabilistic relational planning rules. In *KR-2004*, 2004.

[Pistore and Traverso, 2001] Marco Pistore and Paolo Traverso. Planning as model checking for extended goals in non-deterministic domains. In *ICAPS-01*, 2001.

[Reddy and Tadepalli, 1997] C. Reddy and P. Tadepalli. Learning goal-decomposition rules using exercises. In *ICML-97*, 1997.

[Ruby and Kibler, 1991] D. Ruby and D. F. Kibler. SteppingStone: An empirical and analytic evaluation. In *AAAI-91*. Morgan Kaufmann, 1991.

[Smart and Kaelbling, 2002] William D. Smart and Leslie Pack Kaelbling. Effective reinforcement learning for mobile robotics. In *Proceedings of the International Conference on Robotics and Automation*, 2002.

[Sutton and Barto, 1998] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning*. MIT Press, 1998.

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

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