Learning to Do HTN Planning

Okhtay Ilghami and **Dana S. Nau**
Department of Computer Science
and Institute for Systems Research
University of Maryland
College Park, MD 20742-3255
USA
{okhtay, nau}@cs.umd.edu

Héctor Muñoz-Avila
Department of Computer Science and Engineering
Lehigh University
Bethlehem, PA 18015
USA
munoz@cse.lehigh.edu

Abstract

We describe the *HDL* algorithm, which learns HTN domain representations by examining plan traces produced by an expert problem-solver. Prior work on learning HTN methods required everything to be given in advance except for the methods' preconditions, and the learner would learn the preconditions. In contrast, HDL has no prior information about the methods.

In our experiments, in most cases HDL converged fully with no more than about 200 plan traces. Furthermore, even when HDL was only halfway to convergence, it usually was able to produce HTN methods that were sufficient to solve more than 3/4 of the planning problems in the test set.

Introduction

A big obstacle to the development of planning systems for practical applications is the difficulty of obtaining domain-specific knowledge to help guide the planner. Such information is essential to provide a satisfactory level of performance (e.g., speed of planning, speed of convergence, percentage of problems solved correctly, and so forth), but it can be difficult to get domain experts to spare enough time to provide information detailed and accurate enough to be useful, and it can be even harder to encode this information in the language the planner uses to represent its input. Consequently, it is important to develop ways to learn the necessary domain-specific information automatically. In this paper, we present an eager and incremental learning process for learning such information that is supervised by an expert who solves instances of the problems in that domain.

This paper focuses specifically on Hierarchical Task Network (HTN) planning, in which a planner formulates a plan by recursively applying *methods* that decompose *tasks* (symbolic representations of activities to be performed) into smaller subtasks until *primitive* tasks are reached that can be performed directly by applying classical planning operators.

This paper presents a new algorithm called HDL (HTN Domain Learner) which overcomes some significant limitations of previous work on learning HTN domains:

- Previous work on learning HTN domains (Ilghami *et al.* 2005) required all information about the methods except for their preconditions to be given to the learner in advance, so that the only thing for the learner to learn was the methods' preconditions. In contrast, HDL starts with no prior information about the methods.
- In (Ilghami *et al.* 2005), each input plan trace (i.e., the tree representing the method instances used to decompose an initial task list all the way down to primitive tasks) needed to contain a lot of information so that the learner could learn from it: At each decomposition point in a plan trace, the learner needed to have *all* the applicable method instances, rather than just the one that was actually used. HDL does not need most of that information. At each decomposition point, it only needs to know about one or at most two methods: The method that was actually used to decompose the corresponding task, and one (if there are any) of the methods that matched that task but whose preconditions failed (to serve as a negative training sample).

We present experimental results that reveal the speed with which HDL converges in different situations. In most cases HDL needed no more than about 200 plan traces to converge fully, and in some cases it needed only about 70 plan traces. Furthermore, in every case that we examined, even when HDL was only halfway to convergence, it was able to produce HTN methods that were sufficient to solve about 3/4 of the problems in the test set.

Algorithm

The pseudo-code of HDL is given in Figure 1. Its subroutines are as follows:

- ComputeState(O,S,Π,loc) computes the world state after applying the first loc operators of the plan trace Π with the initial world state S using the operators in O.
- ExtractRelevant(S, I) where S is a normalized[1] state and I is a task list. It returns predicates in a given normalized state that can potentially be relevant to the particular instance of a method currently being considered.[2]
- CE(v,S,b) is an implementation of the Candidate Elimination algorithm on the version space v with training sam-

[1] HDL requires the same notion of normalization that was used in (Ilghami *et al.* 2005). Normalization replaces constants in different training examples that play the same role with one variable in order to generalize the input facts.

[2] See (Ilghami *et al.* 2005) for more details.

Given: O, the set of operators in an HTN domain $D = (T, M, O)$
I = $\{I_1, \ldots, I_n\}$, a set of task lists
S = $\{S_1, \ldots, S_n\}$, a set of world states
Π = $\{\Pi_1, \ldots, \Pi_n\}$, a set of plan traces, one per HTN planning problem (I_i,S_i,D)
Returns: M, a set of learned methods
VS, a set of version spaces each of which represents the precondition of one of the learned methods

```
HDL(O,I,S,Π)
  M = ∅, VS = ∅
  FOR each plan trace Π_i ∈ Π
    FOR each non-leaf or failure node n in Π_i
      Let loc be the number of leaf nodes that occur before n
      S'_i = ComputeState(O,S_i,Π_i,loc)
      IF n is an non-leaf node that decomposes a non-primitive task NT to a task list (t_1,...,t_k)
        Let θ^-1 be the normalizer for this decomposition
        IF there is no method that decomposes (NT)θ^-1 to (t_1,...,t_k)θ^-1 THEN
          Create a new method m that does this decomposition
          Initialize a new version space v that corresponds to m
          M = M ⋃ {m}, VS = VS ⋃ {v}
        FOR each applicable method in n with version space v and normalizer θ^-1
          CE(v,ExtractRelevant(Normalize(S'_i,θ^-1),I), TRUE)
        FOR each inapplicable method in n with version space v and normalizer θ^-1
          CE(v,ExtractRelevant(Normalize(S'_i,θ^-1),I), FALSE)
  RemoveDeadMethods(O,M,VS)
  RemoveRedundantPreconditions(O,M,VS)
  RETURN {M, VS}
```

Figure 1: The HDL Algorithm

ple S. The third argument is a boolean variable determining wether S is a positive or negative training sample.

- RemoveDeadMethods(O,M,VS) removes methods that are dead (i.e., methods that can never be used to solve any planning problem because some of the preconditions of the nodes in the decomposition tree produced by their application can not be satisfied). Note that since each method's version space can potentially represent more than one candidate for what that method's precondition can be, this function should try all the possible combinations of related methods before labeling a method as dead.
- RemoveRedundantPreconditions(O,M,VS) removes all the redundant preconditions of methods (i.e., the preconditions that are always verified in the lower levels of the task hierarchy and therefore are not needed in the higher levels). Since each method's version space can represent more than one candidate for that method's precondition, this function should try all the possible combinations of related methods before labeling a precondition as redundant.

HDL starts by initializing the set of known methods M to an empty set. Then, for each non-leaf node n of each given plan trace, it checks if the decomposition associated with n is already presented by a known method. If not, HDL creates a new method and initializes a new version space to represent its precondition. Then, positive and negative training samples are extracted from non-leaf or failure nodes in each plan trace, and corresponding version spaces are updated. Note that in the normal case, each such node must provide us with one positive and one negative training sample. There are two exceptions to this: First, in some cases there is no negative training sample associated with a node (i.e., all the methods are applicable, thus giving HDL potentially several positive and no negative training samples). Second, in case of failure nodes, there is no positive training sample associated with the node (i.e., none of the methods are applicable, thus giving HDL potentially several negative and no positive training samples). After updating the appropriate version space for all training samples, the dead methods and redundant preconditions are removed in that order (The order is important since the removal of dead methods can produce redundant preconditions).

Empirical Evaluation

There always exists a finite training set that causes HDL to converge to a single domain consistent with the training set. In this Section, we discuss our experiments to figure out how many plan traces are needed to converge on average.

Two domains are used in our experiments: The first one is the blocks world, where our HTN implementation, with 6 operators and 11 methods, is based on the block-stacking algorithm discussed in (Gupta & Nau 1992). The second domain is a simplified version of Noncombatant Evacuation Operation (NEO) planning (Muñoz-Avila *et al.* 1999), with our HTN implementation of 4 operators and 17 methods.

NEO domains are usually complicated, requiring many plan traces to learn them. It is difficult to obtain these plan traces, and even if we had access to the real world NEO plan traces, human experts would need to classify those traces and assess the correctness of the concepts learned by HDL, a very time-consuming process. To overcome this problem, we decided to *simulate* a human expert. We used a correct HTN planner to generate planning traces for random problems in our domains. Then we fed these plan traces to HDL

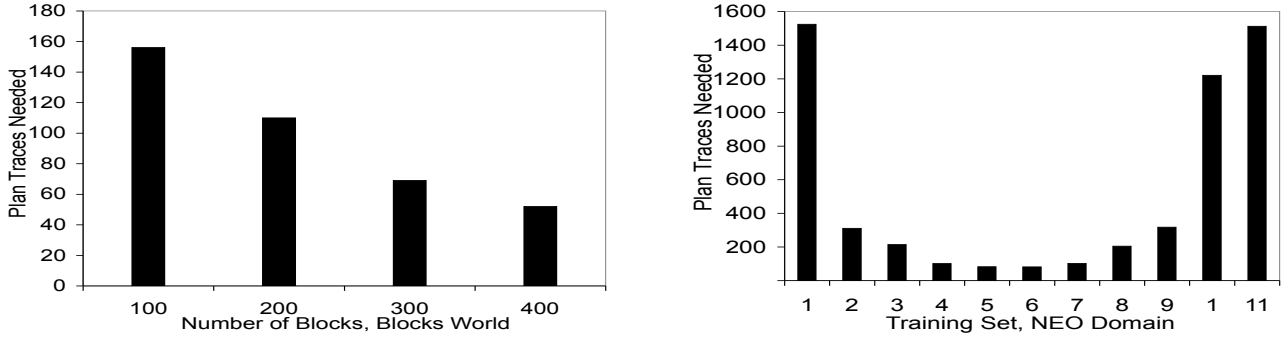

Figure 2: Number of plan traces needed to converge in Blocks world(left) and NEO domain (right).

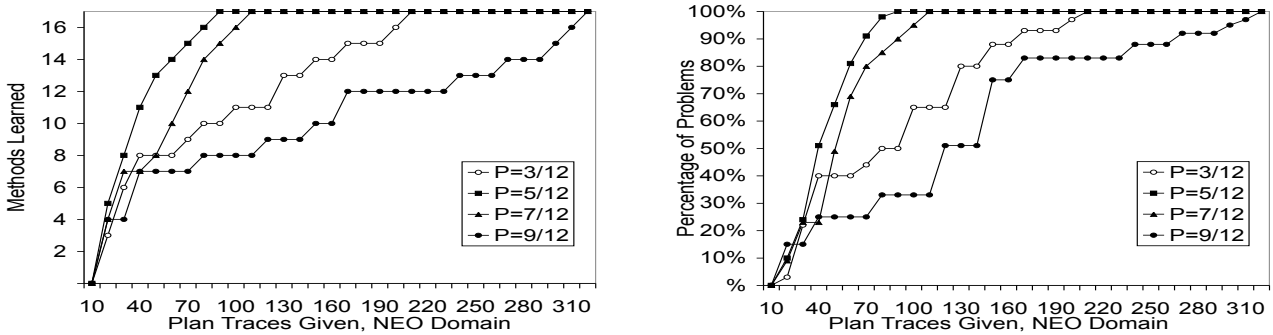

Figure 3: Speed of convergence for different training sets in NEO domain (left), and percentage of problems SHOP could solve using the methods that had already converged (right). The total number of methods in the domain is 17.

and observed its behavior until it converged to the set of methods used to generate these plan traces.

The HTN planner we used is a slightly modified version of SHOP (Nau *et al.* 1999). SHOP, whenever more than one method is applicable, *always* chooses the method that appears first in its knowledge base. Since in our framework there is no ordering on the set of methods, we changed this behavior so that SHOP chooses one of the applicable methods randomly at each point. We also changed the output of SHOP from a simple plan to a plan trace.

Generating the Training Set

For blocks world, we generated 4 sets of random problems with 100, 200, 300, and 400 blocks. We generated initial and goal states block by block, putting each new block randomly and uniformly onto an existing clear block or on the table.

To generate a random NEO problem, every possible state atom was assigned a random variable, indicating whether or not it should be present in the initial world state (e.g., whether there should be an airport in a specific city), or what value its corresponding state atom should have (e.g., whether the evacuation is to take place in a *hostile*, *neutral*, or *permissive* environment). Our preliminary experiments suggested that the values of most variables did not significantly affect the number of plan traces needed to converge. The one exception was the variable indicating if there is an airport in a city. Therefore, we decided to assign a uniform distribution to all random variables other than this variable, and to perform experiments with several different values of this variable, to which we will refer as P in this Section. We conducted 11 sets of experiments, with $P = \frac{1}{12}, \frac{2}{12}, \ldots, \frac{11}{12}$.

Results

The results in this Section are calculated by averaging the results of 10 different randomly-generated training sets. There was little variation in the results of each individual run: The difference between each such result and the average of the results (reported here) was never more than 10%.

Figure 2 compares the number of plan traces HDL needed to converge in our two domains. The number of plan traces needed by HDL to converge in Blocks World decreases as larger problems are given as training samples (because more information is provided in the plan trace of a larger problem). On the other hand, the number of plan traces needed to converge in NEO domain is minimized when the probability P of a city having an airport is approximately 50% (i.e., the 6th training set). When P is close to 0 (i.e., 1st training set), the hard-to-learn methods are those whose preconditions require cities to have airports, because the cases where these methods are applicable somewhere in given plan traces are so rare that the learner cannot easily induce their preconditions. When P is close to 1, the methods that are hard to learn are the ones that do not require cities to have airports. The probability that there is an airport whenever these methods are applicable is high. As a result, the learner cannot induce that an airport's presence is not required.

The first graph in Figure 3 shows the number of methods fully learned as a function of the number of input plan traces when $P = \frac{3}{12}, \frac{5}{12}, \frac{7}{12}, \frac{9}{12}$. When P is close to 0, methods that do not use airports are more likely to be used in the training set, which makes them easier to learn. If we believe that the problems in the test set follow the same distribution as the problems in the training set, it can be argued that these

quickly-learned methods are the most useful ones and therefore the most important to solve the problems in the test set. The same argument can be made when P is close to 1 with the methods that require cities to have airports.

To test our hypothesis that HDL learns the more useful methods faster, we conducted another experiment. In this experiment, HDL tries to solve problems in the NEO domain *before* complete convergence is achieved. In order to solve the problems, HDL used only methods whose preconditions were fully learned, omitting all other methods from the domain. The results of this experiment when $P = \frac{3}{12}, \frac{5}{12}, \frac{7}{12}, \frac{9}{12}$ are shown in the second graph in Figure 3. In the cases where learning was slower, i.e. $P = \frac{3}{12}, \frac{9}{12}$, almost 90% and 75% of problems can be solved respectively using less than 150 plan traces although with 150 plan traces there are respectively 3 and 7 methods out of a total of 17 yet to be fully learned, and learning the entire domain requires more than 200 plan traces in one case and 300 in another (see first graph in Figure 3). This suggests that the more useful methods have already been learned and the extra plan traces are needed only to learn methods that are not as common. It also suggests that HDL can be useful in solving a lot of problems long before all methods are learned.

Related Work

HDL uses version spaces to learn domains. There are, however, other techniques, such as Inductive Logic Programming (ILP) that have been used before to learn hierarchical domain knowledge. Reddy & Tadepalli (1997) introduce X-Learn, a system that uses a generalize-and-test algorithm based on ILP to learn goal-decomposition rules. These (potentially recursive) rules are 3-tuples that tell the planner how to decompose a goal into a sequence of subgoals in a given world state, and therefore are functionally similar to methods in our HTN domains. X-learn's training data consists of solutions to the planning problems ordered in an increasing order of difficulty (authors refer to this training set as an *exercise set*, as opposed to an *example set* which is a set of random training samples without any particular order). This simple-to-hard order in the training set is based on the observation that simple planning problems are often subproblems of harder problems and therefore learning how to solve simpler problems will potentially be useful in solving more difficult ones. Langley & Rogers (2004) describes how ICARUS, a cognitive architecture that stores its knowledge of the world in two hierarchical categories of *concept memory* and *skill memory*, can learn these hierarchies by observing problem solving in sample domains. Garland, Ryall, & Rich (2001) use a technique called *programming by demonstration* to build a system in which a domain expert performs a task by executing actions and then reviews and annotates a log of the actions. This information is then used to learn hierarchical task models. KnoMic (van Lent & Laird 1999) is a learning-by-observation system that extracts knowledge from observations of an expert performing a task and generalizes this knowledge to a hierarchy of rules. These rules are then used by an agent to perform the same task.

Conclusion and Future Work

HDL is an algorithm that learns HTN domains so that they can later be used to solve problems in those domains. Its input consists of plan traces produced by domain experts as solutions to HTN planning problems. Given enough plan traces as input, HDL can learn an HTN domain that is equivalent to the one that presumably was used by the expert.

In our experiments in the NEO domain, HDL learned the most useful methods relatively quickly. This enabled it to solve many planning problems before it had fully learned all of the methods, using only the ones it had fully learned and ignoring the others. For example, after only about half the traces needed for full convergence, HDL produced methods capable of solving about 2/3 of the problems in our test sets.

HDL is incremental: It starts to learn approximations of the methods long before it fully learns any of them. In many cases it should be possible, instead of ignoring the partially-learned methods, to use these approximations to solve planning problems. We believe that by making appropriate modifications to both HDL and the HTN planner, it will be possible to start using HDL's output almost immediately. Our future work will include developing techniques to do this.

Acknowledgments

This work was supported in part by ISLE subcontract 0508268818 and NRL subcontract ??????????? to DARPA's Transfer Learning program, UC Berkeley subcontract SA451832441 to DARPA's REAL program, and NSF grant IIS0412812. The opinions in this paper are ours and do not necessarily reflect the opinions of the funders.

References

Garland, A.; Ryall, K.; and Rich, C. 2001. Learning hierarchical task models by defining and refining examples. In *Proceedings of the 1st Int'l Conference on Knowledge Capture*, 44–51.

Gupta, N., and Nau, D. 1992. On the complexity of blocks-world planning. *Artificial Intelligence* 56(2-3):223–254.

Ilghami, O.; Nau, D. S.; Muñoz-Avila, H.; and Aha, D. W. 2005. Learning preconditions for planning from plan traces and HTN structure. *Computational Intelligence* 21(4):388–413.

Langley, P., and Rogers, S. 2004. Cumulative learning of hierarchical skills. In *Proceedings of the 3rd International Conference on Development and Learning*.

Muñoz-Avila, H.; McFarlane, D.; Aha, D. W.; Ballas, J.; Breslow, L. A.; and Nau, D. S. 1999. Using guidelines to constrain interactive case-based HTN planning. In *Proceedings of the 3rd International Conference on Case-Based Reasoning and Development*, 288–302.

Nau, D. S.; Cao, Y.; Lotem, A.; and Muñoz-Avila, H. 1999. SHOP: Simple hierarchical ordered planner. In *Proceedings of the 16th International Joint Conference on Artificial Intelligence*, 968–973.

Reddy, C., and Tadepalli, P. 1997. Learning goal-decomposition rules using exercises. In *Proceedings of the 14th Int'l Conference on Machine Learning*, 278–286.

van Lent, M., and Laird, J. 1999. Learning hierarchical performance knowledge by observation. In *Proceedings of the 16th Int'l Conference on Machine Learning*, 229–238.