From Plan Traces to Hierarchical Task Networks Using Reinforcements: A Preliminary Report

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

1 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 systems in order to learn 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 *et al.*, 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].

Over the years, researchers have developed several systems that are capable of learning HTNs. Some concentrate on learning applicability conditions of given HTNs such as CaMEL [Ilghami *et al.*, 2005] and DiNCAD [Xu and Munoz-Avila, 2005], while others learn both preconditions and the task decompositions – i.e., the hierarchical structure relating tasks and subtasks. Examples of this approach include X-Learn [Reddy and Tadepalli, 1997], Icarus [Langley and Choi, 2006], LIGHT [Nejati *et al.*, 2006], and our previous work on HTN-Maker [Hogg *et al.*, 2008]. These works on learning hierarchies elicit a hierarchy from a collection of plans or from given action models or from both. It has been demonstrated, both theoretically and experimentally, that existing works usually generate "good" planning knowledge as HTNs. However, it is also known from those evaluations that sometimes the learned HTNs are overly general and in other times overly specific. Overly general methods yield poor planning performance when they are used, while overly specific ones result in too many HTN methods being required in order to cover all or most situations in a planning domain.

This paper describes our work in progress to formalize the "goodness" of the HTN learning. More specifically, we describe the preliminaries for a new HTN Learning approach, which uses our previous work on HTN-Maker as a basis to learn the structures of HTNs and incorporates Reinforcement Learning [Sutton and Barto, 1998] techniques in order to learn "utilities" of those HTNs. This framework computes the utilities of tasks and the methods for them and chooses the best method or methods for each task given as input for the plannig domain. The result is that inferior methods for a task are eliminated, leaving only the best methods to be returned as a solution to the input HTN learning problem.

2 Preliminaries

We use the usual definitions for HTN planning as in Chapter 11 of [Ghallab *et al.*, 2004] and adapt the formalism in our previous work on HTN-Maker [Hogg *et al.*, 2008]. 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 $(n\ arg_1\ \ldots\ arg_k)$ 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, Pre, Del, and 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 to s is a new state $s' = \mathsf{APPLY}(s, a) = (s \setminus \mathsf{Del}) \cup \mathsf{Add}$. A *plan* p is a sequence of actions.

---

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

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

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

In HTN planning, a task is simply a statement with no semantics other than those provided by the methods that decompose it (see below). We define an *annotated task* to be of the form $t = (n, \mathsf{Pre}, \mathsf{Effects})$ where n is a task, Pre is a set of atoms known as the preconditions, and $\mathsf{Effects}$ is a set of atoms known as the *effects*. In this way, a nonprimitive task with annotations may be thought of as an abstract action.

The preconditions and effects associated with an abstract action as above give semantics for accomplishing the annotated task that action specifies and enable us to define an equivalence between an annotated task and a set of goals, and furthermore between a classical planning problem and an HTN planning problem. In particular, given a set of goal atoms g, we define the *equivalent annotated task* as $t = (n, \emptyset, g)$ for some task n.

We restrict ourselves to the Ordered Task Decomposition formalism of HTN planning [Nau *et al.*, 1999]. In this formalism, an *HTN method skeleton* is a pair $m = (h, \mathsf{Subtasks})$, where h is a nonprimitive task (the *head* of the method skeleton) and $\mathsf{Subtasks}$ is a *totally-ordered* sequence of *subtasks*.

An *HTN method* is a triple $m = (h, \mathsf{Pre}, \mathsf{Subtasks})$, where h is a nonprimitive task (the *head* of the method), 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 $\mathsf{Subtasks}$.

We define the notion of an *HTN decomposition* as follows. Given a task t, an *HTN decomposition* for t consists of a plan p that accomplishes t and a subset of the methods from the input domain description that, when successively applied to t and its subtasks, generates the plan p.

Let t be a nonprimitive task and M is a set of methods. Suppose $m \in M$ is a method that can be applied to t (i.e., t is the head of m). Let S_m denote all of the states in which m is applicable to t. In other words, each state $s \in S_m$ satisfies the preconditions of m. Note that we do not need to represent S_m explicitly and exhaustively; the precondition formula of m compactly represents S_m.

The *task-execution function* δ of applying m to t in a state s is given as follows. If $s \notin S_m$ then δ is not defined. Otherwise, $\delta(s, t, m)$ is the set of states defined as follows. For every HTN decomposition for t given M, let p be the plan to be generated by that decomposition and let s' be the state generated by successively applying the actions in p in s. Then, s' is in $\delta(s, t, m)$.

We define the *successor tasks* of applying a method m to t

in s as the set of tasks as follows:

$$\Delta(t, m) = \{t' \mid s \in S_m, s' \in \delta(s, t, m), \text{ and } s' \text{ satisfies}$$
$$\text{the preconditions of the annotated task } t'\}$$

Intuitively, the successor tasks are a set of tasks that could possibly be accomplished following a complete decomposition of the given task from any state.

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 P^h is a plan (i.e., a sequence of actions) p that, when executed in the initial state, performs the desired initial tasks T.

Let $\mathcal{S}$ be the set of all states and $\mathcal{A}$ be the set of all possible actions (i.e., all possible ground instances of the planning operators) in a planning domain. Then, the reward function is defined as $R : \mathcal{S} \times \mathcal{A} \to \Re$. We let γ be the discount factor and α be the learning rate as in [Sutton and Barto, 1998].

We extend the definition of a policy as follows. Let $\mathcal{M}$ be the set of all possible HTN methods in a domain. Then, a *policy* π is a mapping such that $\pi : \mathcal{S} \to \mathcal{A} \cup \mathcal{M}$. A policy π is *defined* for a state s if π specifies an action or a method for s. Otherwise, it is undefined for s. A policy π is *primitive* if for every state s in which π is defined, $\pi(s)$ is an action. Similarly, π is non-primitive if there exists at least one state s for which π is defined and $\pi(s)$ is an HTN method.

Let t be an annotated task and M_t be a set of method skeletons for t. Then, we define the utility of a task and of applying $m \in M_t$ to t using a variant of Q-Learning update rule:

$$U(t) = \max_{m \in M_t} Q(t, m) \qquad (1)$$
$$Q(t, m) = (1 - \alpha)Q(t, m) + \qquad (2)$$
$$\alpha \left[\gamma \max_{t' \in \Delta(t,m)} U(t') - Q(t, m) \right]$$

Intuitively, the utility value of a method skeleton m for an annotated task t provides a measure of "goodness" for decomposing t into the subtasks specified in m.

We define a *learning example* as the tuple (s, g, p), where s is the initial state, g is a goal state, and p is a plan that starts from s and ends in g. In other words, a learning example is a *classical planning problem* along with a solution plan for it.

3 Learning HTNs using Reinforcements

In this section, we describe an incremental, bottom-up learning procedure that starts from a learning example and successively produces HTN methods that form a hierarchy. We call this procedure MaxHTN.

Let A be the set of actions in a planning domain. Let M be the set of HTN methods to be learned, which is the empty set initially. We assume that the utility values for primitive actions are given as input. Given a learning example (s, g, p), the learning procedure iterates over two successive phases. At each iteration, the first phase considers all potential HTN methods from the learning example. [1] The second phase

---

[1]Learning all possible HTN methods from a learning example

immediately follows and produces a (possibly) non-primitive policy in this particular iteration over the learned HTNs M and the actions A of the planning domain. After MaxHTN learns a policy in this iteration, it continues with the next iteration. When there are no new methods that can be generated, the learning procedure terminates and returns the methods in M along with their Q values. We describe these phases below.

Let i denote MaxHTN's current iteration and let L_i be the learning example in this iteration. Initially, we have $i = 0$ and L_0 is the primitive learning example. The first phase, in which MaxHTN generates a set M_i of HTN methods, is very similar to the HTN-Maker learning algorithm. We summarize it here; for details please see [Hogg *et al.*, 2008]. For an input initial state and a solution plan p for a classical planning problem, HTN-MAKER first generates a list S of states by applying the actions in p starting from the initial state s_0. The algorithm then traverses the states in which the effects of an annotated task (i.e., an abstract action) might become true, the states from which the accomplishment of those effects might begin, and the abstract actions whose effects might have been accomplished over that interval. This traversal order in HTN-Maker is chosen deliberately to provide the best opportunity for learned methods to subsume each other. Based on this traversal order, HTN-Maker considers all possible groupings of the actions that appear in L_i and attempts to learn a method for each grouping. In particular, for each grouping HTN-MAKER regresses the effects of the annotated task through the plan elements (actions in p 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 *et al.*, 1986], HTN-MAKER regresses goals both horizontally (through the actions) and vertically (up the task hierarchy through previously-learned methods).

MaxHTN also uses the algorithm outlined above in order to generate the set M_i of the learned methods at the iteration i. If there are no new methods learned at the above step (i.e., if $M_i = \emptyset$), then MaxHTN returns the set of methods M learned so far and the Q-function that specifies the conditions under which each method should be used and its utility. Otherwise, MaxHTN continues with learning the utilities for the methods in M_i.

Then, MaxHTN performs a reinforcement learning step, given the set $M_i \cup A_i$. The input to this RL step also includes the initial state and the goal state of L_i. With these inputs, the RL step performs a simple Q-Learning procedure over the action space defined by $M_i \cup A_i$. When the Q-Learning procedure terminates (we use similar ϵ termination criterion as in [Sutton and Barto, 1998]), the result is a (possibly) non-primitive policy π_i of the current iteration.

An explanation of how to use the methods in M_i in Q-Learning as actions is in order. Suppose s is a state generated during the Q-learning process. If $i = 0$ (i.e., we are in the first

iteration) then M_i is the empty set and Q-learning process proceeds over the set of primitive actions A_0. For $i > 0$, we say that a method $m \in M_i$ is applicable in s if (1) the preconditions in m hold in s and (2) all of its subtasks appear in the policy π_{i-1} at the previous iteration. This way, the Q-Learning process learns a Q-value for a method and state pair, if that method is chosen to be included in the π_i when the learning ends.

Having computing the policy π_i, MaxHTN then updates the set M with the methods that appear in π_i. Suppose S_m denotes the set of all states in which a method $m \in M$ is applicable (i.e., its preconditions hold). For each two methods m and m' in M, if both m and m' are for the same annotated task t, $S_m = S_{m'}$, and $Q(t, m) > Q(t, m')$ (which means that $Q(t, m) > Q(tm')$ for each state $s \in S_m$), then we filter out (i.e., remove) the method m' from M. This ends the current iteration i and MaxHTN goes on to the next iteration $i + 1$. In the next iteration, each path p induced by π_i from the initial state s to the goal state g in the input learning problem constitutes a learning example for MaxHTN at iteration $i + 1$. Currently, we choose one of those paths that has the best Bellman back-up utility computed by a variant of Equation 2 in the usual way; however, we are currently investigating whether learning from all such paths yields more optimal results in MaxHTN.

4 Discussion: Hierarchical Reinforcement Learning

Hierarchical Reinforcement Learning (HRL) has been proposed as a successful research direction to alleviate the well-known "curse of dimensionality" in traditional Reinforcement Learning. Here, hierarchical domain-specific control knowledge and architectures have been used to search MDPs more effectively. See [Barto and Mahadevan, 2003] for an excellent survey on the recent advances on HRL. Below, we summarize the techniques in relation to our work.

The MaxHTN procedure has similarities with the use of *options* in Hierarhical Reinforcement Learning (HRL)[Sutton *et al.*, 1999] in that both MaxHTN and HRL with options use Q-Learning type of learning policies with a mixture of primitive actions and high-level ones. In MaxHTN, a high-level action is an HTN method, whereas in HRL with options, it would be a policy (i.e., an option) that may consists of primitive actions or other options. The important difference is, to the best of our knowledge, MaxHTN learns those HTN methods that participate in the hierarchical Q-Learning process, whereas options are hand-coded by experts and given as input to the reinforcement learning procedure.

Parr [Parr, 1998] developed an approach to hierarchically structuring MDP policies called *Hierarchies of Abstract Machines (HAMs)*. This architecture has the same basis as options, as both are derived from the theory of *semi-Markov Decision Processes*. A HAM policy is defined by a collection of stochastic finite-state machines. During the search in the MDP, a HAM policy induces probabilistic state-transitions, hence yielding different outcome states with different utilities (i.e. rewards and costs). A planner that uses HAMs composes those policies into a policy that is a solution for

---

may not be practical in many situations, given the combinatorics involved. We are planning to investigate heuristic approaches to alleviate this issue, while preserving optimality (or near-optimality).

the MDP. To the best of our knowledge, the HAMs in this work must be supplied in advance by the user rather than being learned on-the-fly by an HTN learner, such as MaxHTN, or more precisely, the HTN generation/learning algorithm it borrows from HTN-Maker.

There are also function approximation and aggregation approaches to hierarchical problem-solving in RL settings. Perhaps the best known technique is Diettrich's MAX-Q decompositions [Dieterich, 2000]. These approaches are based on hierarchical abstraction techniques that are somewhat similar to HTN planning. Given an MDP, the hierarchical abstraction of the MDP is analogous to an instance of the decomposition tree that an HTN planner might generate. Again, however, the MAX-Q tree must be given in advance and it is not learned in an incremental and bottom-up fashion as in our focus in this paper.

The bottom-up and incremental HTN learning is also a point in our work that differentiates MaxHTN with the previous work. To the best of our knowledge, all of the HRL approaches proposed so far are designed in as top-down decomposition procedures, which make them very efficient and useful for eliminating redundant parts of the search space in MDPs and constrainting the value function of the solutions they generate. In our work, we are aiming to use RL in order to learn hierarhical structures in the for of HTNs by observing "good" behavior in every level of the hiearchy and propagating upwards not only the value function (as in MAX-Q for example), but also the structural knowledge as well (since the methods learned in the lower levels determine the methods to be learned in the higher levels).

5 Conclusions

We have described the preliminaries of our work in progress on combining Reinforcement Learning techniques and symbolic HTN learning. The MaxHTN procedure we described above uses this combination in order to learn utilities of HTN methods produced and to generate a "good" HTN knowledge base for a planning domain. MaxHTN performs a bottom-up, incremental learning procedure towards this objective. At each iteration, it learns new HTN methods given what was learned in the previous iterations. Once the methods are generated, it uses a Q-Learning procedure to associate utilities with those methods. Finally, it chooses the best methods for each state and tasks considered in this iteration. We described a method-filtering rule that performs that choice.

In addition to learning HTNs via combining previous learning approachs and RL, one other advantage of the direction we are pursuing is the Q-value update rule we use above. Note that this rule does not mention the state-space of the underlying planning domain, which provides an *abstraction* methodology to do HTN learning. It also avoids the large search spaces that the traditional Q-Learning approaches would explore as well as the large Q-tables they would be required to keep. In that sense, our update rule serves the same purpose as the HRL approaches described above.

We are in the process of developing the formal theory behind our work described here, which will provide theorems for the soundness, convergence, and the optimality of the learned HTNs. To test this theory, we will implement the procedure described above and perform an extensive experimentation with the benchmark problems domains used for both learning HTNs and HTN planning in the literature.

Finally, note also that the above method filtering rule we described in the paper is a very tight and strong one. Although we believe that it is theoretically appropriate and useful, in many practical situations, MaxHTN may not simply filter any methods at any iteration. One way to alleviate this issue is to develop a Pareto-optimal rule in order to filter methods. An alternative, and perhaps more practical solution, would be require a threshold on the percentage of the cases where a method for the same task has higher utilities than another one. In either case, we will focus on *not* introducing a state-based Q-value update rule. We will investigate both in the near future.

References

[Barto and Mahadevan, 2003] Andrew G. Barto and Sridhar Mahadevan. Recent advances in hierarchical reinforcement learning. *Discrete Event Dynamic Systems*, 13(4):341–379, 2003.

[Dieterich, 2000] Thomas G. Dieterich. Hierarchical reinforcement learning with the MAXQ value function decomposition. *JAIR*, 13:227–303, 2000.

[Erol *et al.*, 1996] Kutluhan Erol, James Hendler, and Dana S. Nau. Complexity results for hierarchical task-network planning. *Annals of Mathematics and Artificial Intelligence*, 18:69–93, 1996.

[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 *Proceedings of the 23rd AAAI Conference on Artificial Intelligence (AAAI-08)*, pages 950–956. AAAI Press, 2008.

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

[Langley and Choi, 2006] Pat Langley and Dongkyu Choi. Learning recursive control programs from problem solving. *J. Mach. Learn. Res.*, 7:493–518, 2006.

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

[Nau *et al.*, 1999] D. Nau, Y. Cao, A. Lotem, and H. Munoz-Avila. SHOP: Simple hierarchical ordered planner. In *Proc. of IJCAI '99*, pages 968–973. AAAI Press, 1999.

[Nau *et al.*, 2005] Dana S. Nau, Tsz-Chiu Au, Okhtay Ilghami, Ugur Kuter, Hector Munoz-Avila, J. William Murdock, Dan Wu, and Fusun Yaman. Applications of SHOP and SHOP2. *IEEE Intelligent Systems*, 20(2):34–41, 2005.

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

[Parr, 1998] R. Parr. Hierarchical control and learning for markov decision processes, 1998.

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

[Sacerdoti, 1975] Earl D. Sacerdoti. The nonlinear nature of plans. In *Proc. of IJCAI '75*, pages 206–214, 1975.

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

[Sutton *et al.*, 1999] R. S. Sutton, D. Precup, and S. Singh. Between mdps and semi-mdps: A framework for temporal abstraction in reinforcement learning. *Artificial Intelligence*, 112:181–211, 1999.

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