Learning Methods to Generate Good Plans: Integrating HTN Learning and Reinforcement Learning

Chad Hogg
Dept. of Computer Sci. & Eng.
Lehigh University
Bethlehem, PA 18015, USA
cmh204@lehigh.edu

Ugur Kuter
Institute for Advanced Computer Studies
University of Maryland
College Park, MD 20742, USA
ukuter@cs.umd.edu

Héctor Muñoz-Avila
Dept. of Computer Sci. & Eng.
Lehigh University
Bethlehem, PA 18015, USA
hem4@lehigh.edu

Abstract

We consider how to learn Hierarchical Task Networks (HTNs) for planning problems in which both the quality of solution plans generated by the HTNs and the speed at which those plans are found is important. We describe an integration of HTN Learning with Reinforcement Learning to both learn methods by analyzing semantic annotations on tasks and to produce estimates of the expected values of the learned methods by performing Monte Carlo updates. We performed an experiment in which plan quality was inversely related to plan length. In two planning domains, we evaluated the planning performance of the learned methods in comparison to two state-of-the-art satisficing classical planners, FASTFORWARD and SGPLAN6, and one optimal planner, HSP^*_F. The results demonstrate that a greedy HTN planner using the learned methods was able to generate higher quality solutions than SGPLAN6 in both domains and FASTFORWARD in one. Our planner, FASTFORWARD, and SGPLAN6 ran in similar time, while HSP^*_F was exponentially slower.

Introduction

We consider the problem of learning Hierarchical Task Network (HTN) methods that can be used to generate high-quality plans. HTNs are one of the best-known approaches for modeling expressive planning knowledge for complex environments. There are many AI planning applications that require such rich models of planning knowledge in order to generate plans with some measure of quality.

Learning HTNs means eliciting the hierarchical structure relating tasks and subtasks from a collection of plan traces. Existing works on learning hierarchical planning knowledge describe ways to learn a set of HTN decomposition methods for solving planning problems from a collection of plans and a given action model, but they do not consider the quality of the plans that could be generated from them (Hogg, Muñoz-Avila, and Kuter 2008; Nejati, Langley, and Könik 2006; Reddy and Tadepalli 1997).

Generating optimal plans is a difficult problem; there are an infinite number of possible solutions to planning problems in many domains. Because searching the entire space of plans is intractable, it is desirable to design an algorithm to quickly find plans that are of high-quality but not always optimal.

In this paper, we describe a new synergistic integration of two AI learning paradigms, HTN Learning and Reinforcement Learning (RL). This integration aims to learn HTN methods and their values, which can be used to generate high-quality plans. Reinforcement Learning is an artificial intelligence technique in which an agent learns, through interaction with its environment, which decisions to make to maximize some long-term reward (Sutton and Barto 1998). *Hierarchical Reinforcement Learning* (HRL) has emerged over recent years (Parr 1998; Dietterich 2000) as a formal theory for using hierarchical knowledge structures in RL. However, existing HRL techniques do not learn HTN methods nor their values; they assume a hierarchical structure is given as input.

Our contributions are as follows:

- We describe an RL formalism in which an agent that selects methods to decompose tasks is rewarded for producing high-quality plans.
- We describe how to learn methods and their values based on this formalism. Our approach consists of three related algorithms: (1) our new HTN Learning algorithm, called Q-MAKER, that learns HTN methods by analyzing plan traces and task semantics and computes initial estimates of the values of those methods; (2) an RL algorithm, called Q-REINFORCE, that refines the values associated with methods by performing Monte Carlo updates; and (3) a version of the SHOP planner, called Q-SHOP, that makes greedy selections based on these method values. To the best of our knowledge, this is the first formalism developed for the purpose of integrating these two well-known research paradigms.
- In our experiments, our planner Q-SHOP using methods learned by Q-MAKER and refined by Q-REINFORCE was usually able to generate plans of higher quality than those produced by two satisficing classical planners, FASTFORWARD and SGPLAN6. The time required by our planner was within an order of magnitude of the time used by these satisficing planners. In many cases, our planner produced optimal plans while executing exponentially faster than a planner, HSP^*_F, that guarantees optimality.

Preliminaries

We use the standard definitions for states, planning operators, actions, and classical planning problems as in classical planning (Ghallab, Nau, and Traverso 2004), and tasks, annotated tasks, task networks, methods, and ordered task decompositions in HTN Planning and Learning as in the HTN-MAKER system (Hogg, Muñoz-Avila, and Kuter 2008). We summarize these definitions below.

Classical Planning

A *state* s is a set of ground atoms. An *action* a has a name, preconditions, and positive and negative effects. An action a is *applicable* to a state s if the preconditions of a hold in s. The *result* of applying a to s is a new state s' that is s with the negative effects of a removed and the positive effects added. This relationship is formally described in a *state-transition function* $\gamma : \mathcal{S} \times \mathcal{A} \rightarrow \mathcal{S}$. A *plan* $\pi = \langle a_0, a_1, \ldots, a_n \rangle$ is a sequence of actions. A plan is *applicable* to a state s if the preconditions of a_0 are applicable in s, a_1 is applicable in $\gamma(s, a_0)$, and so forth. The result of applying a plan π to a state s is a new state $\mathsf{apply}(s, \pi) = \gamma(\gamma(\ldots(\gamma(s, a_0), a_1)\ldots), a_n)$.

A *classical planning problem* $P = (\mathcal{S}, \mathcal{A}, \gamma, s_0, g)$ consists of a finite set of states, a finite set of actions, a state-transition function, an initial state, and a goal formula. A *solution* to a classical planning problem P is a plan π such that π is applicable in s_0 and the goal formula holds in $apply(s_0, \pi)$.

HTN Planning & Learning

A *task* t is a symbolic representation of an activity in the world. A task is either *primitive*, in which case it corresponds to the name of an action, or is *nonprimitive*. A *task network* $w = \langle t_0, t_1, \ldots, t_n \rangle$ is an ordered sequence of tasks. A *method* m consists of a name, preconditions, and subtasks. The name is a nonprimitive task. The subtasks are a task network.

A method is *applicable* in a state s to a task t if the method's name is t and the method's preconditions hold in s. The result of applying a method to a task t is called a *reduction* of t using m, and consists of the state s and the subtasks of m. The result of reducing a task t with method m and recursively reducing any nonprimitive subtasks of m with other methods until only primitive tasks remain is a *decomposition* of t with m, $\mathsf{decomp}(t, m) = \pi$.

An *HTN planning problem* $P = (\mathcal{S}, \mathcal{A}, \gamma, \mathcal{T}, \mathcal{M}, s_0, w_0)$ consists of a finite set of states, a finite set of actions, a state-transition function, a finite set of tasks, a finite set of methods, an initial state, and an initial task network. A *solution* to an HTN planning problem is a plan π that is a decomposition of the initial task network and that is applicable in the initial state.

An *annotated task* t is a task with associated preconditions and postconditions that specify what it means to accomplish that task. Using annotated tasks, we can define an equivalence between a classical planning problem and an HTN planning problem.

A Model For Learning Valued Methods

The usual process for solving an HTN planning problem begins with the initial state and initial task network. It proceeds by recursively reducing each of the tasks in the initial task network. When a primitive task is reached, it is applied to the current state and appended to an initially empty plan. When a nonprimitive task is reached, a decision must be made regarding which applicable method to choose or that task and the current state. If the decomposition cannot be completed, this choice becomes a backtracking point.

We model the problem of selecting an applicable method m to reduce a nonprimitive task t as a Reinforcement Learning problem. Thus, the agent is the procedure that makes this choice. When the agent chooses method m to reduce task t it receives a numeric *reward* from the environment, denoted $r(t, m)$. Traditionally, RL uses a slightly different terminology than we adopt here. A *state* in RL literature refers to a situation in which the agent must make a decision, which in our case is the current task to be decomposed. An *action* in RL literature refers to the selection made by the agent, which in our case is the method used to reduce the current task. In this paper, state and action have the same meaning as in the planning literature.

Decomposing nonprimitive tasks is an episodic problem; an episode ends when a plan consisting of only primitive tasks is reached. The objective of the agent is to maximize the total reward received within an episode. This total reward is called the *return*, and is written $R(\mathsf{decomp}(t, m))$. Because this problem is episodic, future rewards do not need to be discounted and the return is simply the sum of all rewards earned in the rest of the episode.

To assist in making informed decisions, the agent maintains a *method-value function* $Q : (\mathcal{T} \times \mathcal{M}) \rightarrow \mathbb{R}$. The value of $Q(t, m)$, called the *Q-value* of m, represents the expected return generated by beginning a decomposition of t by reducing it with m. If the Q-value estimates are accurate, the agent can maximize its likelihood of receiving a high return by selecting the method with highest Q-value among all methods applicable in the current state.

The reason that our method-value function does not specify the value of reducing task t with method m while in state s is that there are typically very many possible states in a planning domain. Thus, it would be impractical to compute state-specific values for each method. Instead, our system implicitly uses the state, since the agent is only capable of selecting among those methods that are applicable in the current state. This abstracts away those features of the state that are not directly relevant to this reduction and makes storing a method-value function feasible.

We use Monte Carlo techniques for solving the RL problem, which means that at the end of an episode the agent updates its method-value function to reflect those rewards that it actually received from the environment. The Q-value of a method is calculated as a function of the returns that agent has actually received when using it in the past. The update formula for $Q(t, m)$ is shown in Equation 1:

$$Q(t, m) \leftarrow Q(t, m) + \alpha(R(\mathsf{decomp}(t, m)) - Q(t, m)) \quad (1)$$

The term α is referred to as the step-size parameter, and can be tuned to give more or less weight to more recent experiences. We maintain a *method-count function* $k : (\mathcal{T} \times \mathcal{M}) \rightarrow \mathbb{Z}^+$ and set $\alpha = \frac{1}{k(t,m)+1}$. This value of α weights older and more recent experiences equally, and thus the Q-value of a method is the average of the returns received when using it.

A *learning example* $e = (s_0, \pi)$ consists of a state s_0 and a plan π that is applicable in that state. We define a *valued HTN learning problem* as a tuple $L = (\mathcal{S}, \mathcal{A}, \gamma, \mathcal{T}, E)$, where $\mathcal{S}$ is a finite set of states, $\mathcal{A}$ is a finite set of actions, γ is the state-transition function, $\mathcal{T}$ is a finite set of annotated tasks, and E is a finite set of learning examples. A *solution* to a valued HTN learning problem is a set of methods $\mathcal{M}$ and a method-value function Q such that:

1. Any solution generated by a sound HTN planner using the methods in $\mathcal{M}$ to an HTN planning problem with an equivalent classical planning problem is also a solution to the equivalent classical planning problem.
2. The equivalent HTN planning problem to every classical planning problem in the domain can be solved by a sound HTN planner using the methods in $\mathcal{M}$.
3. If a sound HTN planner that always selects the method with highest Q-value among applicable methods generates a solution to an HTN planning problem, there does not exist any other solution that produces a higher return.

Note that even if the rewards returned by the environment are predictable (that is, the reward for reducing a given task t with a given method m is always the same), finding a solution to a valued HTN learning problem is very difficult and in some cases impossible. Because we abstract away the states, it is likely that for some two methods m_1 and m_2 there exists states s_1 and s_2 such that m_1 and m_2 are each applicable in both s_1 and s_2 and using m_1 while in state s_1 will result in greater returns than using m_2 in that state, while the reverse is true for s_2. Therefore, we attempt to find approximate solutions to valued HTN learning problems in which the first two conditions hold and the agent will receive good but not necessarily optimal returns.

Solving Valued HTN Learning Problems

Our approach to solving valued HTN learning problems has three phases:

- *Learning methods and initial Q-values*. This phase learns HTN methods while simultaneously obtaining initial estimates for the Q-values from the provided learning examples. The learning examples are analyzed to find action sequences that accomplish an annotated task, producing methods in a bottom-up fashion. The initial Q-values of these methods are calculated from the rewards produced by the decompositions that are built bottom-up from the learning examples, using Equation 1.
- *Refining Q-values*. In this phase no new methods are learned. Instead, the Q-values of the methods are updated through Monte Carlo techniques. Beginning from an arbitrary state and task network in the domain, our Monte Carlo algorithm produces a decomposition top-down, then updates the Q-values of the methods used as in Equation 1. Unlike the first phase, the agent *explores* the method-selection space.
- *Planning with Q-values*. The final phase *exploits* the information captured in the Q-values of methods to find solutions to HTN planning problems that maximize returns.

Q-MAKER: Learning Methods & Initial Q-Values

The Q-MAKER procedure extends the HTN learning algorithm HTN-MAKER (Hogg, Muñoz-Avila, and Kuter 2008) to compute initial estimates of the Q-values of the methods that it learns. Specifically, Q-MAKER takes as input a valued HTN learning problem $L = (\mathcal{S}, \mathcal{A}, \gamma, \mathcal{T}, E)$ and generates a set of methods $\mathcal{M}$, a method-value function Q, and a method-count function k as a solution for L.

HTN-MAKER iterates over each subplan π' of π. Before processing any segment of the plan, it first processes each subplan of that segment. It analyzes each subplan to determine if any annotated tasks have been accomplished by that subplan. If so, it uses hierarchical goal regression to determine how the annotated task was accomplished and creates a method that encapsulates that way of accomplishing the task. When a plan segment accomplishes an annotated task, that annotated task may be used as a subtask when creating methods for any plan segment of which it is a subplan. In this way, decomposition trees are created bottom-up.

In addition to learning the structure and preconditions of these methods, Q-MAKER learns initial Q-values and maintains a count of the number of times a method has been observed. When a new method is learned, Q-MAKER simulates decomposition using that method and its descendants in the learning example and uses the returns as the initial Q-value. When a method had already been learned but is observed again from a new plan segment or entirely different learning example, the initial estimate of the Q-value is updated to include the new scenario using Equation 1 and the counter for that method is incremented. Thus, the initial estimate of the Q-value of a method that was observed several times is the average return from those decompositions using it that appear in the learning examples.

Q-REINFORCE: Refining Q-Values

In the second phase, we refine the initial Q-value estimates using Monte Carlo methods. The RL component is necessary because the initial Q-value estimates may be biased toward the particular learning examples from which the methods were learned. In practice, it might be possible to use those methods in ways that did not appear in the learning examples and that result in very different returns.

Algorithm 1 shows the Q-REINFORCE procedure. Q-REINFORCE recursively reduces an initial task network. At each decision point, Q-REINFORCE selects randomly among the applicable methods, so that the Q-values of all methods have an opportunity to be updated. This makes Q-REINFORCE an *off-policy* learner, since it follows a random-selection policy while improving a greedy-selection policy. Thus, it has an opportunity to explore portions of the

Algorithm 1: The input includes an HTN planning problem $P = (\mathcal{S}, \mathcal{A}, \gamma, \mathcal{T}, \mathcal{M}, s_0, w_0)$, a method-value function Q, and a method-count function k. The output is a plan π, the total return received from decomposing the tasks in w_0, and updated method-value function Q and method-count function k.

```
1  Procedure Q-REINFORCE($P, Q, k$)
2  begin
3    $\pi \leftarrow \langle\rangle$ ; $s \leftarrow s_0$ ; $R = 0$
4    for $t \in w_0$ do
5      if $t$ is primitive then
6        randomly select action $a \in \mathcal{A}$ that matches $t$
7        if $a$ is applicable to $s$ then
8          $s \leftarrow \gamma(s, a)$ ; $\pi \leftarrow \pi \cdot \langle a \rangle$
9        else return FAIL
10     else
11       Let $\mathcal{M}' \subseteq \mathcal{M}$ be those methods for $t$ applicable in $s$
12       if $\mathcal{M}' \neq \emptyset$ then
13         randomly select $m \in \mathcal{M}'$
14         $P' \leftarrow (\mathcal{S}, \mathcal{A}, \gamma, \mathcal{T}, \mathcal{M}, s, \mathsf{Subtasks}(m))$
15         $(\pi', R', Q, k) \leftarrow$ Q-REINFORCE($P', Q, k$)
16         $Q(t, m) \leftarrow Q(t, m) + \alpha(R' + r(t, m) - Q(t, m))$
17         $k(t, m) \leftarrow k(t, m) + 1$
18         $s \leftarrow \mathsf{apply}(s, \pi')$ ; $\pi \leftarrow \pi \cdot \pi'$
19         $R \leftarrow R + R' + r(t, m)$
20       else return FAIL
21   return $(\pi, R, Q, k)$
22 end
```

method-selection space that could not be observed in the learning examples.

From an initially empty plan, current state, and 0 total return (Line 3), the algorithm processes each task in the task network in order (Line 4). Primitive tasks advance the state and are appended to the current plan (Lines 8), but do not involve a decision by the agent and do not produce a reward. For a nonprimitive task, the agent selects an applicable method at random (Line 13). The Q-REINFORCE algorithm is recursively called to decompose the subtasks of the selected method (Line 15). This provides the return from any further reductions of the subtasks of the method, which when summed with the reward of this method selection is the return that the agent receives. The Q-value of the selected method is updated with this return using Equation 1 (Line 16), and the plan and state are suitably updated. The return from this selection is added to a counter of total return to be used in a higher recursive call, if one exists (Line 19).

Q-SHOP: Planning With Q-Values

We also wrote an HTN planner, Q-SHOP, that incorporates our agent for method-selection. The Q-SHOP algorithm is an extension of the SHOP planning algorithm (Nau et al. 1999). While in SHOP the first method encountered is selected, in Q-SHOP our agent selects the method with highest Q-value. In our implementation Q-SHOP does not update Q-values based on its experience, but it could easily be modified to do so.

Experiments

In these experiments we attempted to use our Reinforcement Learning formalism to find plans of minimal length. Thus, when the agent reduces a task t with a method m the reward it receives is the number of primitive subtasks of m multiplied by -1. In this scenario, maximizing returns minimizes the length of the plan generated.

We performed experiments in two well-known planning domains, BLOCKS-WORLD and SATELLITE, comparing the Q-SHOP algorithm versus two satisficing planners, FASTFORWARD and SGPLAN6, and one optimal planner, HSP_F^*. HSP_F^* was the runner-up in the sequential optimization track of the most recent International Planning Competition (IPC-2008), while SGPLAN6 and FASTFORWARD were among the best participants in the sequential satisficing track. The source code and data files for these experiments are available at `http://www.cse.lehigh.edu/InSyTe/HTN-MAKER/`.

Our experimental hypothesis was that using the methods and method-value function learned by Q-MAKER and refined by Q-REINFORCE, Q-SHOP would produce plans that are shorter than those produced by FASTFORWARD and SGPLAN6 in comparable time. We also compared against SHOP using methods learned by Q-MAKER but without the method-value function.

Within each domain, we randomly generated 600 training and 600 tuning problems of varying sizes. First we ran Q-MAKER on solutions to the training problems to produce a set of methods and initial estimate of the method-value function. We refer to results from planning with the methods while ignoring the method-value function as "No QVal", and planning with the methods and initial estimate of the method-value function as "Phase 1". Then we used Q-REINFORCE to solve the 600 tuning problems, updating the Q-values of the methods learned earlier. We refer to results from planning with the methods and the refined method-value function as "Phase 1+2".

For each domain and each of several problem sizes, we then randomly generated 20 testing problems. We attempted to solve each testing problem using the three classical planners, as well as SHOP using the "No QVal" methods and Q-SHOP using the methods and the "Phase 1" and "Phase 1+2" method-value functions. FASTFORWARD, SGPLAN6, SHOP, and Q-SHOP solved the problems in less than a second, while the optimal planner HSP_F^* was much slower. In fact, for problems of even modest size, HSP_F^* was unable to find a solution within a 30-minute time limit.

Figure 1 shows the average plan length generated by the satisficing planners as a percentage of the optimal plan length at each problem size in the BLOCKS-WORLD domain. That is, a value of 100 means that an optimal plan was generated, while 200 means the plan generated was twice as long as an optimal plan. Of these planners, Q-SHOP using the "Phase 1+2" method-value function performs best. In fact, in all but 3 problems out of 420, it found an optimal plan. Next best was Q-SHOP using the "Phase 1" method-value function, which produced plans an average of 13.3%

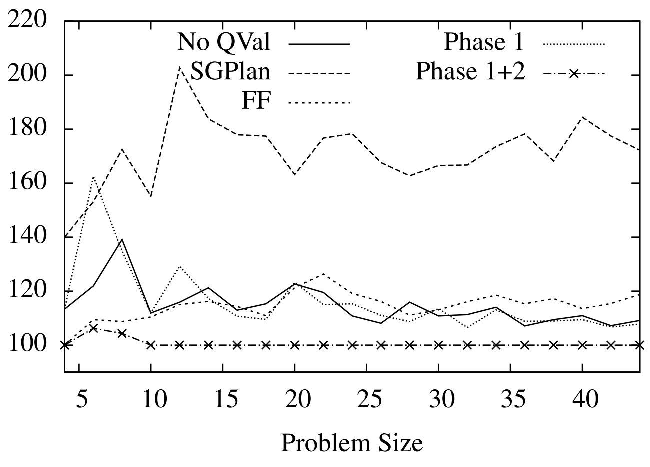

Figure 1: Average Percent Of Shortest Plan Length By Problem Size In BLOCKS-WORLD Domain

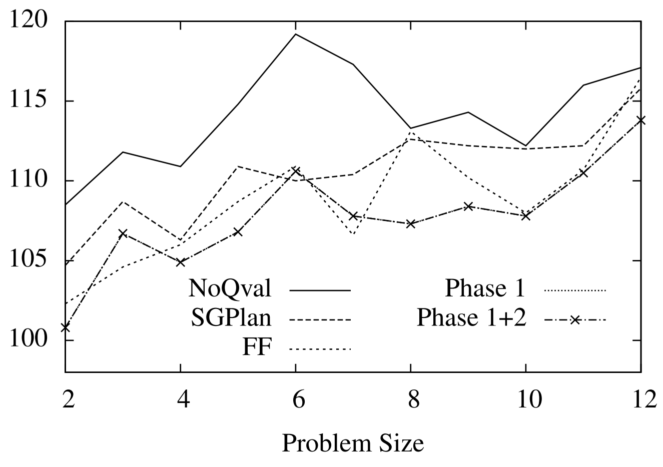

Figure 2: Average Percent Of Shortest Plan Length By Problem Size In SATELLITE Domain

longer than optimal. Third best was SHOP with no Q-values (13.6% longer), then FASTFORWARD (16.1% longer), then SGPLAN6 (73.7% longer). None appeared to be becoming more suboptimal on larger problems than small ones; in fact the opposite was the case for SHOP and Q-SHOP.

The same data for the SATELLITE domain is shown in Figure 2. These results are much closer, but Q-SHOP remains the best choice, producing plans on average 7.6% longer than optimal. In this domain, Q-SHOP produced the same quality plans whether the Q-values were refined or not. Each of the planners is trending toward plans that are further from optimal as the difficulty of the problems increases. We believe that the reason our algorithm performed better in BLOCKS-WORLD than SATELLITE is due to the task representations used. Because Q-SHOP is an Ordered Task Decomposition planner, it is restricted to accomplishing the tasks in the initial task network in the order given. In BLOCKS-WORLD there was a natural ordering that we used, but in SATELLITE this was not the case and the ordering frequently made generating an optimal plan impossible.

Related Work

None of the existing algorithms for HTN Learning attempt to learn methods with consideration of the quality of the plans that could be generated from them. Rather, they aim at finding any correct solution as quickly as possible. These works include HTN-MAKER (Hogg, Muñoz-Avila, and Kuter 2008), which learns HTN methods based on semantically-annotated tasks; LIGHT (Nejati, Langley, and Könik 2006), which learns hierarchical skills based on a collection of concepts represented as Horn clauses; X-LEARN (Reddy and Tadepalli 1997), which learns hierarchical d-rules using a bootstrapping process in which a teacher provides carefully chosen examples; CAMEL (Ilghami et al. 2005), which learns the preconditions of methods given their structure; and DINCAT (Xu and Muñoz-Avila 2005), which uses a different approach to learn preconditions of methods given their structure and an ontology of types.

The QUALITY system (Pérez 1996) was one of the first to attempt to learn to produce high-quality plans. QUALITY produced PRODIGY control rules from explanations of why one plan was of higher quality than another. These rules could be further processed into trees containing the estimated costs of various choices. At each decision point while planning, PRODIGY could reason with these trees to select the choice with lowest estimated cost.

In this paper, we described an integration of HTN Learning with Reinforcement Learning to produce estimates of the costs of using the methods by performing Monte Carlo updates. As opposed to other RL methods such as Q-learning, MC does not bootstrap (i.e., make estimates based on previous estimates). Although in some situations learning and reusing knowledge during the same episode allows the learner to complete the episode more quickly, this is not the case in our framework; RL is used to improve the quality of plans generated but does not affect satisfiability.

Hierarchical Reinforcement Learning (HRL) has been proposed as a successful research direction to alleviate the well-known "curse of dimensionality" in traditional Reinforcement Learning. Hierarchical domain-specific control knowledge and architectures have been used to speed the Reinforcement Learning process (Shapiro 2001). Barto (2003) has written an excellent survey on recent advances on HRL.

Our Q-MAKER procedure has similarities with the use of *options* in Hierarchical Reinforcement Learning (Sutton, Precup, and Singh. 1999). In Q-MAKER, a high-level action is an HTN method, whereas in HRL with options, it would be a policy that may consists of primitive actions or other options. The important difference is that Q-MAKER learns those HTN methods that participate in the Reinforcement Learning process, whereas in the systems with which we are familiar options are hand-coded by experts and given as input to the Reinforcement Learning procedure.

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 planner that uses HAMs composes those policies into a policy that is a solution for the MDP. To the best of our knowledge, the HAMs in this work must be supplied in advance by the user.

There are also function approximation and aggregation approaches to hierarchical problem-solving in RL settings. Perhaps the best known technique is MAX-Q decompositions (Dietterich 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 a bottom-up fashion as in our work.

Conclusions

We have described a new learning paradigm that integrates HTN Learning and Monte Carlo techniques, a form of Reinforcement Learning. Our formalism associates with HTN methods Q-values that estimate the returns that a method-selection agent expects to receive by reducing tasks with those methods. Our work encompasses three algorithms: (1) Q-MAKER, which follows a bottom-up procedure starting from input traces to learn HTN methods and initial estimates for the Q-value associated with these methods, (2) Q-REINFORCE, which follows a top-down HTN plan generation procedure to refine the Q-values, and (3) Q-SHOP, a variant of the HTN planner SHOP that reduces tasks by selecting the applicable methods with the highest Q-values.

We demonstrated that, with an appropriate reward structure, this formalism could be used to generate plans of good quality, as measured by the inverse of plan length. In the BLOCKS-WORLD domain, our experiments have demonstrated that Q-SHOP consistently generated shorter plans than FASTFORWARD and SGPLAN6, two satisficing planners frequently used in the literature for benchmarking purposes. In fact, the Q-SHOP plans were often optimal. In the SATELLITE domain Q-SHOP and FASTFORWARD performed similarly while SGPLAN6 was poorer. We also tested the planner HSP_F^*, which always generates optimal plans, but it runs much more slowly than the other planners and could only solve the simplest problems within the bounds of our hardware.

The addition of Q-values allowed Q-SHOP to find better plans than SHOP, demonstrating that incorporating Reinforcement Learning into the HTN Learning procedure was beneficial. The experiments also demonstrated a synergy between Q-MAKER and Q-REINFORCE: in the BLOCKS-WORLD domain the plans generated with Q-SHOP using the initial estimates learned by the former and refined by the latter were shorter on average than when using only the initial estimates.

In the future, we would like to investigate using our framework for incorporating Reinforcement Learning and HTN Learning with different reward schemes, such as actions with non-uniform cost or dynamic environments such as games, in which the quality of plans generated is not necessarily a direct function of the actions in the plan but could be related to the performance of that plan in a stochastic environment. The Monte Carlo techniques used in our work have the advantage of simplicity, but converge slowly. Thus, we are interested in designing a model in which more sophisticated RL techniques, such as Temporal Difference learning or Q-Learning, could be used instead.

Acknowledgments. This work was supported in part by NGA grant HM1582-10-1-0011 and NSF grant 0642882. The opinions in this paper are those of the authors and do not necessarily reflect the opinions of the funders.

References

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

Dietterich, T. G. 2000. Hierarchical reinforcement learning with the MAXQ value function decomposition. *JAIR* 13:227–303.

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

Hogg, C.; Muñoz-Avila, H.; and Kuter, U. 2008. HTN-MAKER: Learning HTNs with minimal additional knowledge engineering required. In *Proceedings of AAAI-08*.

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

Nau, D.; Cao, Y.; Lotem, A.; and Muñoz-Avila, H. 1999. SHOP: Simple hierarchical ordered planner. In *Proceedings of IJCAI-99*.

Nejati, N.; Langley, P.; and Könik, T. 2006. Learning hierarchical task networks by observation. In *Proceedings of ICML-06*.

Parr, R. 1998. *Hierarchical Control and Learning for Markov Decision Processes*. Ph.D. Dissertation, University of California at Berkeley.

Pérez, M. A. 1996. Representing and learning quality-improving search control knowledge. In *Proceedings of ICML-96*.

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

Shapiro, D. 2001. *Value-Driven Agents*. Ph.D. Dissertation, Stanford University.

Sutton, R. S., and Barto, A. G. 1998. *Reinforcement Learning*. MIT Press.

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

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