Learning Probabilistic Hierarchical Task Networks to Capture User Preferences

Nan Li, Subbarao Kambhampati, and Sungwook Yoon
School of Computing and Informatics
Arizona State University
Tempe, Arizona 85281 USA
nan.li.3@asu.edu, rao@asu.edu, Sungwook.Yoon@asu.edu

Abstract

While much work on learning in planning focused on learning domain physics (i.e., action models), and search control knowledge, little attention has been paid towards learning user preferences on desirable plans. Hierarchical task networks (HTN) are known to provide an effective way to encode user prescriptions about what constitute good plans. However, manual construction of these methods is complex and error prone. In this paper, we propose a novel approach to learning probabilistic hierarchical task networks that capture user preferences by examining user-produced plans given no prior information about the methods (in contrast, most prior work on learning within the HTN framework focused on learning "method preconditions"—i.e., domain physics—assuming that the structure of the methods is given as input). We will show that this problem has close parallels to the problem of probabilistic grammar induction, and describe how grammar induction methods can be adapted to learn task networks. We will empirically demonstrate the effectiveness of our approach by showing that task networks we learn are able to generate plans with a distribution close to the distribution of the user-preferred plans.

1 Introduction

Application of learning techniques to planning is an area of long standing research interest. Most work in this area to-date has however focused on learning either search control knowledge, or domain physics. Another critical piece of knowledge needed for plan synthesis is that of user preferences about desirable plans, and to our knowledge there has not been any work focused on learning it. It has long been understood that users may have complex preferences on plans (c.f. [Baier and McIlraith, 2008]). Perhaps the most popular approach for specifying preferences is by hierarchical task networks (or HTNs), where in addition to the domain physics (in terms of primitive actions and their preconditions and effects), the planner is provided with a set of non-primitive actions (tasks) and methods for reducing them into combinations of primitive and non-primitive actions. Figure 1 shows a set of HTNs for a travel domain. A plan (sequence of primitive actions)

is considered a valid HTN plan if and only if it (a) is executable and achieves the goals and (b) can be produced by reducing non-primitive tasks. While the first clause focuses on goal achievement, the second clause ensures that the plan produced is one that satisfies the user preferences.

For the example in Figure 1, as specified, the top level goal of traveling from a source to a destination can be achieved either by *Gobytrain*, which involves a specific sequence of tasks, or *Gobybus*. In contrast, the plan of hitch-hiking from the source to the destination, while executable, is not considered a valid plan. The reduction schemas can be viewed as providing a "grammar" of desirable solutions, and the planner's job is to find executable plans that are also grammatically correct.

While HTNs can be used to specify the grammar of user-desired solutions, manual construction of the HTNs is complex and error prone. In this paper, we focus on learning this grammar, given only successful plans known to be acceptable to the users. Our approach takes off from the accepted view of task reduction schemas as specifying the grammar of desirable solutions[Geib and Steedman, 2007; Kambhampati *et al.*, 1998]. We extend this understanding in two ways: First, we consider weighted task reduction schemas. That is, each reduction schema for a goal is associated with a probability p specifying the users preference for that particular reduction. This is a useful generalization as we can now capture the degree of user's preference for a specific plan (instead of just a binary preference judgement).[1] Second, we exploit the connection between reduction schemas and grammar by adapting the considerable work on grammar induction [Collins, 1997; Charniak, 2000; Lari and Young, 1990]. Specifically, we view the sample plan traces as sentences generated by a target grammar of schemas, and develop an expectation-maximization (EM) algorithm for learning task reduction schemas for that grammar.

We emphasize that our focus is only on capturing user preferences, and not on learning about feasibility. User preferences need not be based on feasibility; indeed a user having preferences based on feasibility is akin to the fox in Aesop's fable of Sour Grapes. A preferred plan may thus not necessarily be executable. In the travel example, a plan to take the train will be a preferred one, but may not be executable if there is no train station. It is the responsibility of the planner

---

[1]Note that it is trivial to get non-weighted reduction schemas from weighted ones, if so desired—just keep all schemas whose weights are over a certain threshold and ignore their weights.
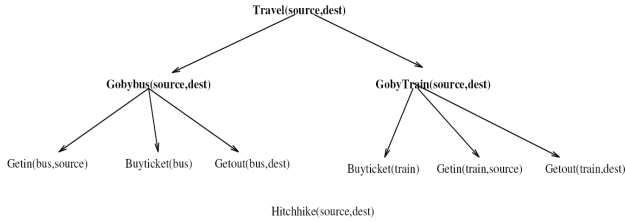
Figure 1: Hierarchical task networks in a travel domain.

Table 1: Ordered probabilistic hierarchical task networks (in Chomsky normal form) in the travel domain

Primitive actions: $Buyticket, Getin, Getout, Hitchhike$;
Non-primitive actions: $Travel, A_1, A_2, A_3, B_1, B_2$;
$Travel \rightarrow 0.2, \quad A_2 \ B_1$
$Travel \rightarrow 0.8, \quad A_1 \ B_2$
$B_1 \rightarrow 1.0, \quad A_1 \ A_3$
$B_2 \rightarrow 1.0, \quad A_2 \ A_3$
$A_1 \rightarrow 1.0, \quad Buyticket$
$A_2 \rightarrow 1.0, \quad Getin$
$A_3 \rightarrow 1.0, \quad Getout$

to ensure that the most preferred executable plan is returned [Baier and McIlraith, 2008]. (It could however be possible to combine preference and feasibility learning schemes; see Section 5 for a discussion.)

In the following sections, we start by formally stating the problem of learning probabilistic hierarchical task networks (pHTNs). Next, we discuss the relations between probabilistic grammar induction and pHTN learning. After that, we present an algorithm that acquires pHTNs from example plan traces. The algorithm works in two phases. The first phase hypothesizes a set of schemas that can cover the training examples, and the second is an expectation maximization phase that refines the probabilities associated with the schemas. We then evaluate the effectiveness of our approach by comparing the distributions of user-desired plans and the plans produced from our learned task networks. We conclude with a discussion of the related work and a summary of our contributions.

2 Probabilistic Hierarchical Task Networks

We define a pHTN domain H, as a 3-tuple, $H = \langle A, NA, S \rangle$, where A is a set of primitive actions, NA is a set of non-primitive actions, and S is a set of reduction schemas indexed by non-primitive actions. We follow the normal STRIPS semantics for the primitive actions. Each non-primitive action $na_i \in NA$ is associated with a set of reduction schemas. Each reduction schema s_j can be seen as a 3-tuple, $\langle na_i \rightarrow p, dec \rangle$, where dec is an ordered list of primitive and non-primitive actions, and p is the probability of choosing that decomposition. This probability specifies the preference of the user. Without loss of generality, we restrict our attention to reduction schemas in the Chomsky normal form, with each schema decomposing a non-primitive task to either two non-primitive tasks or a single primitive task. For example, for the Travel domain presented in Figure 1, Table 1 shows an example of the pHTN decomposition rules. According to these, the user prefers using train (80%) to using bus (20%).

A problem R for a pHTN domain H is a pair of states $R = \langle I, G \rangle$, I is the initial state, G is the partial description of the desired goal state. A primitive action sequence o is a valid solution to H and R if o can be executed from I leading to a state where G holds, and there is some reduction process in H that derives o from the top level goal. The probability of $p(o)$ is $\sum_{DEC} \Pi_{dec \in DEC} \ p(dec)$, where DEC is all the decomposition processes that can generate o.

We can now state the pHTN learning problem formally. Given a set $O : o_1, o_2 \cdots o_k$ of training plans (each of which are sequences of primitive actions satisfying the goal), find a set of pHTNs H^l that most likely generates the observed primitive action sequence. Thus $H^l = argmax_H \ p(O \mid H)$.

In this work, we simplify the learning task further by focusing on learning parameter-less schemas. The assumption is reasonable in domains such as the travel domain, but does not work with domains like Blocks World, where the action sequences are differentiated by bindings. We also assume that the user preferences can be expressed as unconditional pHTNs. Thus we disallow reduction schemas of the form, "*if you are in Europe, prefer trains more than planes*."[2] This ensures that pHTNs correspond to context free grammars.

3 Learning pHTNs

It is clear that the pHTN as defined above has strong similarity to probabilistic context free grammars (PCFG). There is a one-to-one correspondence between the non-terminals of PCFG and the HTN non-primitive symbols. Rather than developing pHTN learning algorithm from the scratch, we will exploit existing techniques for PCFG induction. In particular, expectation maximization (EM) is the weapon of choice when it comes to PCFG induction and we shall use it also for learning pHTN [Lari and Young, 1990]. The pHTN learning problem does differ from existing work on PCFG induction in some critical respects. For example, we assume that the input to the algorithm is just a primitive action sequence, without any annotations about non-primitive actions. As discussed in the following section, our algorithm invents non-primitive symbols as needed. We are not aware of any PCFG learning study that directly works in such a setup.

Our algorithm consists of two parts. First, we have a greedy structure hypothesizer, which creates non-primitive symbols, and associated reduction schemas, as needed to cover all the training examples. The key guiding principle here is the parsimonious generation of reduction schemas in Chomsky normal form. In the second phase, an expectation-maximization approach is used to iteratively refine the probabilities of the reduction schemas.

3.1 Greedy Structure Hypothesizer (GSH)

The pseudo code for the GSH algorithm is shown in algorithm 1. GSH learns reduction rules in a bottom-up fashion. It starts by initializing the schema set S to schemas associated with primitive actions. Next the algorithm detects whether there are recursive structures embedded in the plans, and learns a recursive schema for them. Recursive structures are of the form of continuous repetitions of a single terminal/non-terminal action with another termina/non-terminal action appearing once before or after the repetitions,

---

[2] Note that the condition we are talking about is on the preference rather than method applicability/feasibility.

such as $\{a, a, ..., a, b\}$ and $\{a, b, b, ...b\}$. If both the length of the repetitions and the frequency the repetitions appearing in the plans meet the minimum thresholds, a recursive structure is said to be detected. The thresholds are decided by both the average length of the given plans and the total number of plan examples. For instance, in plan $\{\alpha_1, \alpha_2, \alpha_2, \alpha_2, \alpha_3\}$ (where α denotes either a primitive or non-primitive action), $\{\alpha_1, \alpha_2, \alpha_2, \alpha_2\}$ and $\{\alpha_2, \alpha_2, \alpha_2, \alpha_3\}$ are considered as recursive structures. After identifying a recursive structure, the structure learner can construct a recursive schema out of it. Take $\{\alpha_1, \alpha_2, \alpha_2, \alpha_2\}$ as an example, the acquired schema for it would be $\alpha_1 \rightarrow \alpha_1 \ \alpha_2$.

If the algorithm fails to find recursive structures, it starts to search for the action pair that appears in the plans most frequently, and constructs a reduction for the action pair. To build a non-recursive schema, the algorithm will introduce a new symbol and set it as the head of the new schema. After getting the new schema, the system updates the current plan set with this schema by replacing the action pairs in the plans with the head of the schema.

Having acquired all the reduction schemas, the structure learning algorithm assigns initial probabilities for these schemas. Note that for consistency, the sum of probabilities associated with all ways of reducing a non-primitive task must add up to 1. Thus, if there are k reduction schemas with the same head symbol, then each of them are assigned the probability $\frac{1}{k}$. To break ties among reduction schemas with the same head, GSH adds a small random number to each probability and normalizes the values again. This output of GSH is a redundant set of reduction schemas, which is sent to the EM phase.

Example: For example, in a variant of the travel domain, where the traveler can buy a day pass and take the train multiple times, two training plans are shown on the top right in Figure 2. Primitive schemas, $A_1 \rightarrow Buyticket$, $A_2 \rightarrow Getin$, $A_3 \rightarrow Getout$, are first constructed for each action. Updated plans are shown as level 2 in Figure 2. Next, since $A_2 A_3$ is the most frequent action pair in the plans, the structure hypothesizer constructs a rule $S_1 \rightarrow A_2 \ A_3$. After updating the plans with the new rule, the plans become (A_1, S_1) and (A_1, S_1, S_1, S_1) as shown as level 3 in Figure 2. Next, GSH detects a recursive structure in plan (A_1, S_1, S_1, S_1) and learns a rule $A_1 \rightarrow A_1 \ S_1$. At this point, since all of the plans are parsable by existing schemas, GSH stops constructing new rules. All of the rules constructed in this example are shown in the bottom left in Figure 2.

3.2 Refining Schema Probabilities: EM Phase

The probabilities associated with the initial set of schemas generated by the GSH phase are tuned by an expectation-maximization algorithm. Since all plan examples can be generated by the target reduction schemas, each plan should have a parse tree associated with it. However, the tree structures of the example plans T are not provided. Therefore, we consider T as the hidden variables. We will use T(o, H) to denote the parse tree of a plan example o given the reduction schemas H. The algorithm operates iteratively. In each iteration, it involves two steps, an E step, and an M step.

In the E step, the algorithm estimates the values of the hidden variables T, which, in this case, are the tree structures associated with each plan example with symbol g as the root

---

Algorithm 1: GSH constructs an initial set of reduction schemas, S, from the plan examples, O.

Input: Plan Example Set O.
1   S := primitive action reduction schemas;
2   **while** *not-all-plans-are-parsable(O, S)* **do**
3   **if** *has-recursive-schema(O)* **then**
4   s := generate-recursive-schema(O);
5   **else**
6   s := generate-most-frequent-schema(O);
7   **end**
8   S := $S + s$;
9   O := update-plan-set-with-schema(O, S);
10   **end**
11   S = initialize-probabilities(S);
12   **return** S

---

node, denoted as $p(T \mid O, H)$.

To do this, the algorithm computes the most probable parse tree for each plan example. Any subtree of a most probable parse tree is also a most probable parse subtree. Therefore, for each plan example, the algorithm builds the most probable parse tree in a bottom-up fashion until reaching the start symbol g. For the lowest level, since each primitive action only associates with one reduction schema of the form $na \rightarrow a$, the most probable parse trees for them are directly recorded as their only associated primitive reduction schemas. For higher levels, the most probable parse tree is decided by

$$s, i = argmax_{s,i} \ p(s \mid H) * p(T(o_1, H) \mid o_1, H)$$
$$* \ p(T(o_2, H) \mid o_2, H). \tag{1}$$

where o is the current action sequence, $a_1, a_2, \ldots a_n$; s is a reduction schema of the form $a_{root} \rightarrow a_l \ a_r$, which specifies the reduction schema that is used to parse o at the first level; i is an integer between 1 to n, which determines the place that separates o into two subtraces, o_1 and o_2. o_1 is the action sequence, $a_1, a_2, \ldots a_i$, and o_2 is the action sequence, $a_{i+1}, \ldots a_n$. After getting s and i, the most probable parse tree of the current trace consists of a_{root} as the root, and the most probable parse trees for the subtraces, $T(o_1, H)$ and $T(o_2, H)$, as the left and right child of the root. The probability of that parse tree is: $p(s \mid H) * p(T(o_1, H) \mid o_1, H) * p(T(o_2, H) \mid o_2, H)$. This bottom-up process continues until it finds out the most probable parse tree for the entire plan. Note that the algorithm stated above constructs a parse tree even if the probability associated with it is 0. In order to reduce the complexity of the E step, parse trees that depend on reduction schemas with 0 probabilities are directly pruned without calculating the most probable parse subtrees.

After getting the parse trees for all plan examples, the algorithm moves on to the M step. In this step, the algorithm updates the selection probabilities associated with the reduction schemas by maximizing the expected log-likelihood of the joint event

$$H_{n+1} = argmax_{H_n} \Sigma_T \ p(T \mid O, H_n) \log p(O, T \mid H_n) \tag{2}$$

where H_n stands for the probabilities of reduction schemas in the nth iteration. For a reduction schema with head a_i, the new probability of getting chosen is simply the total number

Figure 2: Example illustrating the operation of Greedy Structure Hypothesizer (see text)

of times that schema appearing in the parse trees divided by the total number of times a_i appearing in the parse trees.

After finishing the M step, the algorithm starts a new iteration until convergence. The output of the algorithm is a set of probabilistic reduction schemas.

Discussion: Notice that although the EM step does not introduce new reduction schemas, it deletes redundant reduction schemas by assigning low or zero possibilities to them. We also note that learning preferences from example traces can suffer from overfitting problem: By generating exact reduction schemas for each example plan, we will get the reduction schemas that produce only the training examples. Our greedy schema hypothesizer addresses this issue by detecting recursive schemas to avoid overfitting, and by constructing schemas giving preference to frequent action pairs to reduce the total number of non-primitive actions in schemas.

4 Empirical Evaluation

To evaluate the ability of our approach to learn pHTNs, we designed and carried out experiments in both synthetic and benchmark domains. All the experiments were run on a 2.13 GHz Windows PC with 1.98GB of RAM. Although we focus on learning accuracy rather than cpu time, we should clarify up-front that the cpu time for learning was quite reasonable. It ranged between 0 to 44 milliseconds per domain per training plan.

Evaluation presents special challenges as we need to see whether our algorithm is able to adequately capture user preferences. We avoided costly direct user studies through an oracle-based experimental strategy: we assume access to the ideal pHTN schemas capturing user preferences: H^*. We use H^* to generate training examples which are fed to the learning algorithm. The pHTN schemas learned by our algorithm, H^l are then compared to H^*. The main comparison between the two schema sets is in terms of the distribution of plans they generate. Additionally, we also compare them in terms of number of non-primitive actions used, since redundant schemas may lead to overfitting, and can also slow down the preference computation at runtime.

To compare the distribution of the plans generated by H^* and H^l, we use Kullback-Leibler divergence measure, defined as $D_{KL}(\mathcal{P}_{H^*}||\mathcal{P}_{H^l}) = \Sigma_i \, \mathcal{P}_{H^*}(i) \log \frac{\mathcal{P}_{H^*}(i)}{\mathcal{P}_{H^l}(i)}$, where $\mathcal{P}_{H^l}$ and $\mathcal{P}_{H^*}$ are distributions of plans generated by H^l and H^* respectively. This measure goes to 0 if the distribution

is identical and goes potentially to infinity if the distributions differ significantly.

4.1 Experiments in Randomly Generated Domains

In these experiments, we first randomly generate a set of recursive and non-recursive schemas, and use them as H^*. In non-recursive domains, the randomly generated schemas form a binary and-or tree with the goal as the root. The probabilities for the schemas are also assigned randomly (and normalized so that probabilities of all the schemas with the same head sum to 1). Generating recursive domains is similar with the only difference being that 10% of the schemas generated are recursive. We also varied the size of the given schemas by the number of non-primitive actions. The number of training plans, and the number of testing plans are adjusted accordingly. For instance, if the input schemas contain n non-primitive actions, the number of training plans is *10n*, and the number of testing plans is *100n*. For each schema size, we averaged our results over 100 randomly generated schemas of that size.

Rate of Learning: In order to test the learning speed, we first measured KL divergence values with 15 non-primitive actions given different numbers of training plans. The results are shown in Figure 3(a). We can see that even with small number of training examples, our learning mechanism can still construct pHTN schemas with KL divergence no more than 0.2. As the number of training cases increases, our algorithm learns better schemas. However, the learning rate slows down. Note that we did not report the KL divergence with very small number of training examples. This is because when the training plans provided are not enough to represent the structures embedded in the target schemas, the learned schemas will not be able to generate plans with those uncovered structures. In this case, KL divergence will be infinity.

Effectiveness of the EM Phase: To examine the effect of the EM phase, we carried out experiments comparing the KL divergence between $\mathcal{P}_{H^*}$ and $\mathcal{P}_{H^l}$; as well as the KL divergence between $\mathcal{P}_{H^*}$ and $\mathcal{P}_{H^g}$ (Figure 3(b) and Figure 3(c)), where H^g is the set of schemas generated by the greedy hypothesizer, which are subsequently refined by the EM phase into H^l. Inspection reveals that although KL divergence increases in larger domains, in domains without recursive schemas, KL divergence between the original plan distribution ($\mathcal{P}_{H^*}$) and the learned plan distribution ($\mathcal{P}_{H^l}$) is no more than 0.066 with 50 non-primitive actions. This is much
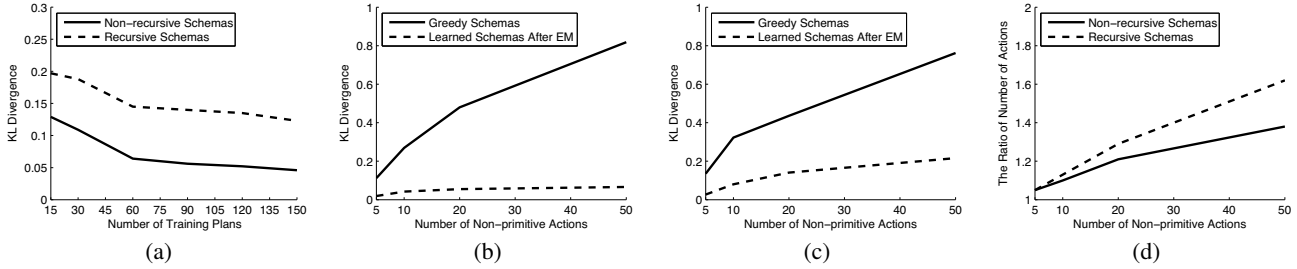
Figure 3: Experimental results in synthetic domains (a) KL Divergence values with different number of training plans. (b) KL Divergence between plans generated by original and learned schemas in *non-recursive* domains. (c) KL Divergence between plans generated by original and learned schemas in *recursive* domains. (d) Measuring conciseness in terms of the ratio between the number of actions in the learned and original schemas.

smaller than the KL divergence 0.818 between the original plan distribution and the plan distribution that would be generated by the schemas output by the GSH phase ($\mathcal{P}_{H^g}$). The EM phase is thus effective in refining H^g to H^l.

Conciseness of Learned Schemas: The conciseness of the schemas is also an essential factor measuring the quality of the schemas, since by ignoring it one can trivially generate schemas with low KL divergence by continuously adding new schemas for each training plan. To measure conciseness, we compute the ratio between the number of actions in the learned schemas (H^l) and the original schemas (H^*). Figure 3(d) presents the results. We can see that with less than 10 non-primitive actions in a domain, the constructed schemas have only 1 or 2 non-primitive actions more than the original schemas. This is acceptable, since even manually constructed schemas may be of different sizes and are usually not the most concise schemas. However, the ratio increases to 1.6 when the original schemas contain 50 non-primitive actions, which would not be considered sufficiently compact in capturing the structure of schemas. Future work in structural learning may be able to alleviate this problem.

Effect of Recursive Schemas: We note that KL divergence for domains with recursive schemas is larger than that for domains without recursive schemas. This is because in domains that contain recursive schemas, the plan space is infinite. The finite number of plans generated by these schemas is not able to represent the exact distribution embedded in the schemas. Even for two sets of plans generated by the same schemas, KL divergence is not zero.

4.2 Benchmark Domains

In addition to the experiments with synthetic domains, we also picked two of the well known benchmark planning domains and developed Chomsky normal form pHTNs for them. Then we generated training plans and evaluated the learning algorithm on them.

Logistics Planning: The domain we used in the first experiment is a variant of the Logistics Planning domain, inside which both planes and trucks are available to move packages. There are 11 non-primitive actions, and 4 primitive actions, *load*, *fly*, *drive* and *unload*, in this domain. We presented 100 training plans to the learning system. The training plans consist of different ways of moving a package. Moreover,

Table 2: Learned schemas in Logistics

Primitive actions: $load, fly, drive, unload$;
Non-primitive actions: $movePackage, S_0, S_1, S_2,$
S_3, S_4, S_5;

$movePackage \rightarrow 0.17,\ movePackage\ movePackage$	
$movePackage \rightarrow 0.25,\ S_0\ S_5$	$S_5 \rightarrow 1.0,\ S_3\ S_2$
$movePackage \rightarrow 0.58,\ S_0\ S_4$	$S_4 \rightarrow 1.0,\ S_1\ S_2$
$S_0 \rightarrow 1.0,\ load$	$S_1 \rightarrow 1.0,\ fly$
$S_2 \rightarrow 1.0,\ unload$	$S_3 \rightarrow 1.0,\ drive$

these training plans show a preference for moving packages by plane over moving by truck, and a preference for using less number of trucks and planes (less steps in the plan).

The KL divergence between the original schemas and the learned schemas in this domain is 0.04. Table 2 shows schemas learned from the training plans. We can see that the generated schemas successfully captured both the structure and the preference in the input plans. The second and third schemas for *movePackage* show that you can move a package either by plane or by truck. The first schema is a recursive case which means that you can repeatedly move the package until reaching the destination.

Gold Miner: The second domain we used is Gold Miner. It is a domain that is used in the learning track of the 2008 International Planning Competition, in which a robot is in a mine and tries to find the gold inside the mine. The robot can pick up bombs or a laser cannon. The laser cannon can destroy both hard and soft rocks, while the bombs can only penetrate soft rocks. Moreover, the laser cannon will also destroy the gold if the robot uses it to uncover the gold location. The desired strategy for this domain is: *1) get the laser cannon, 2) shoot the rock until reaching the cell next to the gold, 3) get a bomb, 4) use the bomb to get gold.*

The training schemas have 12 non-primitive actions and 5 primitive actions, *move*, *getLaserCannon*, *shoot*, *getBomb* and *getGold*. We gave the system 100 plans of various lengths generated by these schemas. Table 3 shows the schemas learned for this domain. The KL divergence between the original and learned schemas in this domain was relatively high at 0.52. This can be explained by the significantly higher recursion in the schemas in this domain. Nevertheless, it is easy to see that the learned schemas do prefer plans that obey the desired strategy, while the number of moves the robot needs

Table 3: Learned schemas in Gold Miner

Primitive actions: $move, getLaserGun, shoot,$
$getBomb, getGold$;
Non-primitive actions: $goal, S_0, S_1, S_2,$
S_3, S_4, S_5, S_6;

$goal \rightarrow 0.78, \ S_0 \ goal$	$goal \rightarrow 0.22, \ S_1 \ S_6$
$S_0 \rightarrow 1.0, \ move$	$S_1 \rightarrow 0.78, \ S_1 \ S_5$
$S_1 \rightarrow 0.22, \ getLaserGun$	$S_5 \rightarrow 1.0, \ S_2 \ S_0$
$S_2 \rightarrow 1.0, \ shoot$	$S_6 \rightarrow 1.0, \ S_3 \ S_4$
$S_3 \rightarrow 0.71, \ S_3 \ S_0$	$S_3 \rightarrow 0.29, \ getBomb$
$S_4 \rightarrow 1.0, \ getGold$	

to get the gold varies in cases. Specifically, the plans sanctioned by the learned schemas start by moving to get the laser cannon, followed by shooting all the rocks using the laser cannon, and finish by using the bomb to get the gold.

5 Discussion and Related Work

In the planning community, HTN planning has for long been given two distinct and sometimes conflicting interpretations (c.f. [Kambhampati *et al.*, 1998]): it can be interpreted either in terms of domain abstraction (with non-primitive actions mediating access to the executable ones) or in terms of user preferences (with HTNs providing a grammar for the solutions desired by the user). While the original top-down HTN planners have been motivated by the former view and aim at higher efficiency than primitive action planning, the latter view has lead to the development of bottom-up HTN planners [Barrett and Weld, 1994], and explains the seeming paradox of higher complexity for HTN planning (afterall, finding *a* plan cannot be harder than finding one that satisfies complex preferences).

Despite this dichotomy, most prior work on learning HTN models (e.g. [Ilghami *et al.*, 2002; Langley and Choi, 2006; Yang *et al.*, 2007; Hogg *et al.*, 2008]) has focused only on the domain abstraction angle. Typical approaches here require the structure of the reduction schemas to be given as input, and focus on learning applicability conditions for the non-primitive tasks. In contrast, our work focuses on learning HTNs as a way to capture user preferences, given only successful plan traces. The difference in focus also explains the difference in evaluation techniques. While most previous HTN learning efforts are evaluated in terms of how close the learned schemas and feasibility conditions are to the actual schemas, we focus on the distribution of plans generated by the learned and original schemas.

An intriguing question is whether pHTNs learned to capture user preferences can, in the long run, be over-loaded with domain semantics. In particular, it would be interesting to combine the two HTN learning strands by sending our learned pHTNs as input to the existing feasibility learners. The applicability conditions that are learned on the non-primitive actions can then be used to allow efficient top-down interpretation of user preferences.

As discussed in [Baier and McIlraith, 2008], besides HTNs, there are other representations, such as trajectory constraints expressed in linear temporal logic, for expressing user preferences. It will be interesting to explore methods for learning preferences in those representations too, and see to what extent common user preferences are naturally expressible in HTNs.

6 Conclusion

Despite significant interest in learning in the context of planning, most prior work focused only on learning domain physics or search control. In this paper, we motivated the need for learning user preferences. Given a set of example plans conforming to user preferences, we developed a framework for learning probabilistic HTNs that are consistent with these examples. Our approach draws from the literature on probabilistic grammar induction. We provided a principled empirical evaluation of our learning techniques both in synthetic and benchmark domains. Our primary empirical evaluation consisted of comparing the distributions of plans generated from the learned schemas and target schemas (presumed to represent the user preferences), and demonstrates the effectiveness of learning.

We are currently extending this work in several directions, including learning parameterized pHTNs, learning conditional preferences, exploiting partial schema knowledge and handling the "feasibility" bias in the training data (which is caused by the fact that we learn only from successful plan traces, and some of the plans preferred by the user may have been filtered out because of infeasibility).

References

[Baier and McIlraith, 2008] Jorge A. Baier and Sheila A. McIlraith. Planning with preferences. *AI Magazine*, 29(4):25–36, 2008.

[Barrett and Weld, 1994] Anthony Barrett, Daniel S. Weld: Task-Decomposition via Plan Parsing. AAAI 1994: 1117-1122

[Charniak, 2000] Eugene Charniak. A maximum-entropy-inspired parser. Proc. *ACL*, 2000.

[Collins, 1997] Michael Collins. Three generative, lexicalised models for statistical parsing. In Proc. *ACL*. 1997.

[Geib and Steedman, 2007] Christopher W. Geib and Mark Steedman. On natural language processing and plan recognition. Proc. *IJCAI*, 2007.

[Hogg *et al.*, 2008] Chad Hogg, Héctor Muñoz-Avila, and Ugur Kuter. Htn-maker: Learning htns with minimal additional knowledge engineering required. Proc. *AAAI*. 2008.

[Ilghami *et al.*, 2002] Okhtay. Ilghami, Dana S. Nau, Hector Muñoz Avila, and David W. Aha. Camel: Learning method preconditions for HTN planning. Proc *AIPS*, 2002.

[Kambhampati *et al.*, 1998] Subbarao Kambhampati, Amol Mali, and Biplav Srivastava. Hybrid planning for partially hierarchical domains. Proc. *AAAI*, 1998.

[Langley and Choi, 2006] Pat Langley and Dongkyu Choi. A unified cognitive architecture for physical agents. Proc. *AAAI*. 2006.

[Lari and Young, 1990] K. Lari and S. J. Young. The estimation of stochastic context-free grammars using the inside-outside algorithm. *Computer Speech and Language*, 4:35–56, 1990.

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