Learning Hierarchical Task Networks by Observation

Negin Nejati NEGIN@STANFORD.EDU
Pat Langley LANGLEY@CSLI.STANFORD.EDU
Tolga Konik KONIK@STANFORD.EDU
Computational Learning Laboratory, Center for the Study of Language and Information, Stanford University, Stanford, CA 94305 USA

Abstract

Knowledge-based planning methods offer benefits over classical techniques, but they are time consuming and costly to construct. There has been research on learning plan knowledge from search, but this can take substantial computer time and may even fail to find solutions on complex tasks. Here we describe another approach that observes sequences of operators taken from expert solutions to problems and learns hierarchical task networks from them. The method has similarities to previous algorithms for explanation-based learning, but differs in its ability to acquire hierarchical structures and in the generality of learned conditions. These increase the method's capability to transfer learned knowledge to other problems and supports the acquisition of recursive procedures. After presenting the learning algorithm, we report experiments that compare its abilities to other techniques on two planning domains. In closing, we review related work and directions for future research.

1. Introduction

In recent years, hierarchical task networks have emerged as a powerful framework for representing and organizing knowledge about action (Ilghami et al., 2002; Wilkins & desJardins, 2001). With access to such content, an agent can generate or execute plans far more effectively than it can from domain operators alone, because the knowledge specifies how to decompose complex tasks into simpler ones. Such approaches have been applied successfully to a variety of challenging domains, and they scale to complexity much better than classical planning methods.

Despite their clear advantages, hierarchical task networks can be difficult and time consuming to construct manually. This suggests the use of machine learning to acquire the knowledge base from experience, but, despite a substantial literature on learning for planning (Zimmerman & Kambhampati, 2003), there have been remarkably few efforts toward acquiring hierarchical plan structures. Moreover, most work in this area has focused on learning from the results of search, which can produce poor solutions and may even fail entirely on some tasks.

An alternative approach is to learn from traces of an expert's behavior. Research on behavioral cloning (e.g., Sammut, 1996) incorporates this idea, but typically learns flat rules for reactive control. Two other paradigms – learning apprentices (e.g., Mitchell et al., 1985) and programming by demonstration (e.g., Cypher, 1993) – emphasize different induction methods and problems but also focus on nonhierarchical structures. In this paper, we adapt the general idea of learning from expert traces to acquire a class of hierarchical task networks – *teleoreactive logic programs* – that are distinctive because they index methods by the goals they achieve. We can state the learning task more precisely in terms of its inputs and outputs:

- *Given*: a set of operators that produce predictable effects under known conditions;
- *Given*: a set of training problems, each of which specifies an initial state and a goal;
- *Given*: for each training problem, a sequence of operator instances that achieves the goal from the initial state;
- *Find*: a teleoreactive logic program that reproduces the solutions to the training problems and generalizes well to new ones.

Appearing in *Proceedings of the 23rd International Conference on Machine Learning*, Pittsburgh, PA, 2006.

We will see that, unlike research on behavioral cloning, our approach chains backward from the goal achieved by each sample trace to analyze the reasons for each action. In this sense, it bears similarity to early work on explanation-based learning, which also relied on goal regression. However, these methods jettisoned the explanation structure after learning, whereas our approach retains it to determine the hierarchical organization of learned knowledge.

In the next section, we describe the knowledge structures given to and acquired by our method, along with the performance mechanisms that operate over that knowledge. After this, we describe the learning algorithm in detail and illustrate its operation on a simple example. Next we report experiments with the technique on two planning domains, including comparisons with more traditional learning methods. Finally, we examine at more length how our approach relates to earlier work on learning plan knowledge and discuss some directions for additional research.

2. Knowledge Representation

Before we can explain our performance and learning methods, we must first describe the representation we use to encode the knowledge they use and acquire. This formalism supports a specialized class of hierarchical task networks that we call *teleoreactive logic programs*, which Langley and Choi (2006) report in more detail. Such programs consist of two distinct but interconnected knowledge bases that describe different aspects of a domain.

The first knowledge base includes a set of concept definitions that recognize classes of situations in the domain and describe them at different levels of abstraction. Concepts are encoded in a hierarchical language and shape the system's beliefs about the domain. Each concept includes a head, which is stated as a predicate with zero or more arguments, and a body, which includes one or more arithmetic tests or relations that must hold. The lowest-level concepts only refer to the perceptual inputs from the domain, while the higher level ones can refer to other concepts as well. Table 1 shows examples of two primitive concept definitions, `on` and `hand-empty`, and two nonprimitive concepts, `clear` and `unstackable`, for the Blocks World.

The second knowledge base contains the definitions of primitive and high-level skills that can be executed to change the environment. Primitive skills are defined in terms of the actions the agent can perform with each clause having a head that specifies its name and arguments, a set of typed variables, a single start condition, a set of effects, and a set of executable actions. The effects of each primitive skill are known to the system and are specified in terms of known concepts. These correspond to *primitive tasks* or *operators* in hierarchical task networks.

Table 1. Examples of concepts from the Blocks World.

```
(on (?blk1 ?blk2)
 :percepts ((block ?blk1 x ?x1 y ?y1)
            (block ?blk2 x ?x2 y ?y2 h ?h))
 :tests    ((equal ?x1 ?x2)
            (>= ?y1 ?y2)
            (<= ?y1 (+ ?y2 ?h))))
(clear (?block)
 :percepts  ((block ?block))
 :negatives ((on ?other?block)))
(unstackable (?block ?from)
 :percepts  ((block ?block) (block ?from))
 :positives ((on ?block ?from)
             (clear ?block) (hand-empty)))
(hand-empty ()
 :percepts  ((hand ?hand status ?status))
 :tests     ((eq ?status 'empty)))
```

Higher level skills, which are the result of the learning, specify ways to decompose complex activities into simple ones. They correspond to *methods* in hierarchical task networks but differ in that the head of each skill clause is the goal concept it achieves if executed to completion. This introduces a connection between concepts and skills that is essential for our learning method, as we describe in Section 4. The body of a high-level skill includes the necessary perceptual inputs, the start conditions, and the relevant subskills. Table 2 and Figure 1 show examples of primitive and high-level skills from the Blocks World.

Both concept and skill knowledge bases are stored in long-term memories, while the specific instances of them are stored in short-term memories that change as the environment evolves.

3. Inference and Execution Mechanisms

The performance mechanisms of a teleoreactive logic program reflect the fact that it operates in a physical setting that changes over time. As Langley and Choi (2006) report, the basic architecture functions in discrete cycles. On each cycle it deposits, in a perceptual buffer, low-level sensory information about objects within the agent's field of view. The system then calls an inference module that operates in a bottom-up, data-driven manner to infer relational beliefs from these perceptions and concept definitions. The inference procedure first produces instances of primi-

Table 2. Two primitive skills for the Blocks World domain.

```
(unstack (?block ?from)
 :percepts ((block ?block ypos ?ypos)
            (block ?from))
 :start    ((unstackable ?block ?from))
 :effects  ((clear ?from)
            (holding ?block))
 :actions  ((*grasp ?block)
            (*v-move ?block (+ ?ypos 10))))
(stack (?block ?to)
 :percepts  ((block ?block)
             (block ?to x ?x y ?y h ?h))
 :start     ((stackable ?block ?to))
 :effects   ((on ?block ?to)
             (hand-empty))
 :actions   ((*h-move ?block ?x)
             (*v-move ?block (+ ?y ?h))
             (*ungrasp ?block)))
```

tive concepts, which depend only on perceived objects, then asserts beliefs based on higher-level concepts.

After this, the architecture invokes an execution module that, in contrast with inference, operates in a top-down manner to achieve high-level goals. On each cycle, using the beliefs produced during inference, the system finds all applicable paths through the skill hierarchy that start from the agent's goal. Since goals are stated as concept instances and the heads of non-primitive skills refer to the concept they achieve, the agent can easily pick executable paths that are relevant to its current goal. The applicability of a skill path depends on the applicability of every skill clause on that path. An individual skill instance is applicable when in the current state its preconditions are satisfied but its head is not satisfied. Note that a skill path is not about a course of action over time; it describes the hierarchical context from the highest-level skill down to a primitive skill. When the agent carries out the actions associated with the selected skill path, the environment changes and the architecture repeats the same procedure until it achieves the goal.

To assure reactivity, on the cycle the system also considers the previously executed subskills. This enables it to invoke those subskills again if their objectives are no longer satisfied due to unexpected events. On the other hand, to prevent the agent from switching among multiple intentions instead of persistently working on one and moving to the next, the architecture prefers skill paths that are more similar to the path that it chose on the previous cycle.

Our approach to skill utilization bears many similarities to those for hierarchical task networks (Wilkins & desJardins, 2001), which have often emphasized plan execution over plan generation. However, most work in this tradition has relied on sophisticated notations for describing complex activities that make them difficult to learn. Our formalism for teleoreactive logic programs is closer to the one assumed by Ilghami et al.'s (2002) SHOP2, which focuses on plan generation rather than execution. As we will see shortly, this representation supports the incremental learning of hierarchical skills from traces of expert behavior.

Table 3. A training example for learning by observation from the Blocks World.

```
Goal: (clear A)
Primitive skill sequence:
  ((unstack C B) (putdown C T1) (unstack B A))
Initial state:
  ((unstackable C B) (hand-empty) (clear C)
   (ontable A T1) (on C B) (on B A))
```

4. Learning Mechanism

Now that we have described the processes that interpret a teleoreactive logic program to achieve an agent's goals, we can turn to methods for learning these structures by observation. As mentioned earlier, our approach builds on knowledge about a domain and expands upon it to incorporate skills that can achieve complicated tasks. This background knowledge includes the definitions for concepts and primitive skills.

Our system learns from problem-solution pairs provided by the expert. Each problem is defined by a goal instance and a specific initial state of the environment. The solution is a sequence of primitive skill instances provided by the expert that achieves the goal starting from the initial state. Table 3 shows a training example for a simple problem in the Blocks World domain. The initial state for this problem has a three-block tower with `C` on `B` and `B` on `A`, and the goal is to get block `A` clear.

Given these inputs, the algorithm proceeds to parse and explain the solution steps using a strategy similar to that in explanation-based learning (Ellman, 1989). It starts by chaining backward from the goal by focusing on the final primitive skill provided by the expert, such as `(unstack B A)` in our example from Table 3. At each step, it selects between two alternative accounts of the expert trace.

If the primitive skill contains the goal as one of its effects, the algorithm explains the goal using *skill chaining*. If this skill is applicable in the initial state of

the problem, one can directly execute it to achieve the goal. However, if its precondition is not satisfied, extra work is needed to make it applicable. In this case, the algorithm tags the single precondition of this skill as its new goal and tries to justify the previous solution steps with respect to it. Figure 1 illustrates the parsing procedure on the Blocks World example from Table 3. In this example, the last primitive skill, `(unstack B A)`, achieves the goal `(clear A)` directly, but its precondition, `(unstackable B A)`, is not satisfied in the initial state. This unsatisfied precondition is considered as the goal the expert was pursuing before the final move. Therefore, the algorithm proceeds to justify the primitive skill sequence `(unstack C B)(putdown C T1)` for achieving `(unstackable B A)`.

When the goal is not one of the effects of the primitive skill in the sequence, the algorithm explains it using *concept chaining*. In this case, the skill either achieves one of the goal's subconcepts or it is an unnecessary step that presumably achieves some other purpose. The system determines the possible subgoals from the definition of the goal concept, and then looks back through the state sequence to find the order in which the expert achieved them. All primitive skills executed before achieving a subgoal are treated as potential contributors. This can lead to considering a skill for more than one subgoal, which is reasonable since skills can have multiple effects that contribute to different aims. In addition, this lets the system handle interleaved subtraces when the expert alternates between pursuing different subgoals. When a subtrace includes a skill that is irrelevant to achieving a goal, the system simply ignores it. Each subgoal with its associated sequence of primitive skills poses a new problem is handled by calling the main routine recursively.

Continuing with our Blocks World example, the sequence `(unstack C B) (putdown C)` in Table 3 makes `B` and `A unstackable`, but none of these primitive skills include `(unstackable B A)` as their effects. This results in concept chaining and breaking down `(unstackable B A)` into its subgoals – `(hand-empty)`, `(on B A)`, and `(clear B)` – as defined in Table 1. By tracking the order of subgoal achievements through the state sequence, `(unstack C B)` is assigned to `(clear B)`, while both `(unstack C B)` and `(putdown C)` are considered as contributors to achieving `(hand-empty)`. The method will detect the literal `(unstack C B)` as an irrelevant step for this goal later in its processing.

After the algorithm has finished parsing the expert's solution trace, it uses the explanation structure to learn new skill clauses for future use. For skill chaining, if the precondition P of the current primitive skill

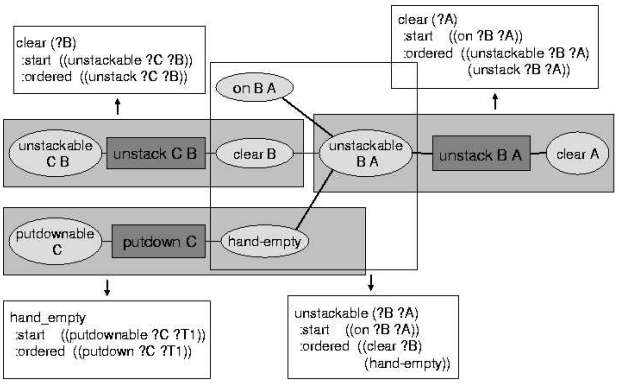

Figure 1. Illustration of the learning algorithm on a simple problem from the Blocks World and the acquired skills.

S is satisfied in the initial state and the expert did nothing to achieve that, the system learns a new skill clause with the achieved goal G as its head, P as its start condition, and a single subskill S.[1] If the expert has done work to achieve P, the system learns a skill clause that achieves G by composing the skill that achieves P (which has the same head as P and has already been learned in the shallower recursion calls) with S. The start conditions of the learned skill are the same as the start conditions of the skill P. In Figure 1, `(clear ?B)` and `(clear ?A)` are examples of skill clauses learned through skill chaining.

For concept chaining, if the goal G has n subgoals, $g_1, \ldots, g_n$, and m of them are satisfied in the initial state, the lower level skills that are learned for achieving $g_{m+1}, \ldots, g_n$ comprise the subskills field of the new skill, but the $g_1, \ldots, g_m$ concepts determine the start condition of this new skill. The nonprimitive skill `(unstackable ?B ?A)` in Figure 1 is an example of a skill clause learned by this procedure. As we discuss shortly, the acquired precondition does not guarantee successful execution, but it still often serves as a useful heuristic. The learning mechanism can produce disjunctive skills, since different methods of achieving a goal have the same head. This convention leads to flexible skills, since each clause can be expanded with all combinations of its subskills' different clauses.

As demonstrated elsewhere (Langley & Choi, 2006), one can use traces generated by a problem solver to learn analogous skills. The method introduced here replaces the problem-solving traces with ones from an expert performing the task. Although we use a similar subroutine for creating skill clauses, the resulting hierarchy can differ because the expert may provide different solution traces than the problem solver. We

[1] For the distinction between this learned clause and the primitive skill it incorporates, see Langley and Choi (2006).

expect that learning by observation will be more practical in complex domains because of the heavy search they impose on problem solvers.

5. Experimental Evaluation

To evaluate the behavior of our approach to learning teleoreactive logic programs by observation, we designed and carried out experiments in two planning domains. Below we describe our experimental design, after which we report the results of these studies.

5.1. Experimental Design

Since our method operates in an incremental manner, we chose to employ an online training regimen in which we presented randomly sampled problems one at a time. If the system fails to achieve the goal using its current knowledge, it retrieves an expert trace stored with the problem. After learning a set of skill clauses from this trace, the system turns to the next problem. This lets us measure performance as a function of the number of problems encountered. We selected the cumulative number of tasks solved successfully as our performance metric, primarily because different problems involved distinct levels of difficulty, and thus were not directly comparable. We also averaged the results across different sequences of problems to guard against order effects. Other metrics such as CPU time are not relevant here, as the performance system does not include a problem solver and simply fails to solve a task if it does not have all the necessary skills.

Because our approach to learning by observation constructs an explanation of the expert trace, it seems appropriate that we compare its behavior to traditional methods for explanation-based learning. We considered two variations on this idea, each tied to one of our theoretical claims. First, we maintain that our method should acquire more flexible knowledge than explanation-based techniques because it retains the explanation structure in its skill hierarchy. Thus, we implemented a standard algorithm for learning a flat macro-operator (Mooney, 1990) from each expert trace that collects a set of sufficient conditions, with constants replaced by variables, to achieve that problem's top-level goal. This macro-operator can solve the training problem on which it is based, but the method finds very specific conditions, and preliminary experiments suggested it generalized so poorly to new tasks that we did not pursue it further.

A second claim revolves around our method for selecting conditions on learned skills, which does not rely on goal regression and only uses information about component skills and concepts. For comparison purposes, we developed an alternative technique that still learns hierarchical skills by analyzing expert traces, but that invokes the same condition-finding scheme for each clause as for macro-operator formation. This produces conditions that are sufficient to achieve the goal in each clause's head, making them more specific than those our method generates, but also less likely to initiate chains of action that the agent cannot complete. Initial studies suggested that this alternative generalized reasonably well, so we included it in our formal experiments.

We settled on two domains for our studies: the Blocks World and Depots, which we describe more fully later. The former is not especially difficult, but it is well understood, and we had a handcrafted task network that we could compare with learned skills. Depots is a more challenging domain that involves a much larger search space, which we predicted would help distinguish our approach from the explanation-based alternative. However, because we had no handcrafted program against which to compare learned skills, we could only evaluate the methods on this domain in terms of their behavior.

5.2. Experiments with the Blocks World

The Blocks World consists of a table, a gripper, and several cubical blocks that are distributed in one or multiple columns on the table. The system distinguishes different situations in the environment using nine defined concepts, including the ones in Table 1. It can also change its environment using four primitive skills: `stack`, `unstack` (see Table 2), `pickup`, and `putdown`. The goals refer to configurations of blocks that are defined via existing concepts. This knowledge is sufficient, in principle, to solve all the problems in the domain by employing extensive search. However, when there are more than a few blocks in the environment, search is slow and impractical.

Our experiment used 240 random problems with three distinct goals: `clear`, `holding`, and `on`. The environment included up to seven blocks with random initial configurations and the difficulty level of the problems ranged from one to 16-step solutions. Inspection revealed the system learned skills that are equivalent to the handcrafted ones. Figure 1 shows the skills acquired from the sample input in Table 3. These high-level skills equip the system to achieve goals like clearing a block on many new problems. One reason is their potential for being applied recursively. For example, the learned skills in the figure can clear a block regardless of the number of blocks on top of

Table 4. Some examples of skills acquired in the Blocks World using explanation-based learning.

```
(clear (?A)
 :percepts ((block ?A)(block ?B)
            (block ?C))
 :start    ((hand-empty)(on ?B ?A)
            (on ?C ?B)(clear ?C)
 :ordered  ((unstackable ?B ?A)
            (unstack ?B ?A)))

(unstackable  (?B ?A)
 :percepts ((block ?A)(block ?B)
            (block ?C))
 :start    ((hand-empty)(on ?B ?A)
            (on ?C ?B)(clear ?C))
 :ordered  ((clear ?B)(hand-empty)))
```

it. Table 4 shows the analogous skills learned by the explanation-based method, which have very specific start conditions. This results in learning distinct versions of `(clear ?A)` for problems with different number of blocks stacked on block `A`.

The left hand graph in Figure 2 shows the superior performance of our system compared to explanation-based learning. The plots show the cumulative number of successful examples averaged over 20 different selections and orderings of 100 input problems in the Blocks World domain for each algorithm. On average, our system learns 14 new skills from a few problems and solves 100 percent of the test problems, while the explanation-based method learns 119 skills and solves fewer than 50 percent of the novel problems. The dotted line shows the behavior of an expert who can solve all problems from the outset. Our method's behaviour follows this line very closely, with a small lag caused by a few failures at the early stages of learning. The small error bars show that the behaviors are similar across different problem orderings.

5.3. Experiments with Depots

Depots is a more complicated domain that was introduced in the Third International Planning Competition (Bacchus, 2000). With crates that can be loaded into trucks and driven to different locations where they are unloaded and stacked onto pallets, it combines attributes of Blocks World with logistics planning. Since a typical problem involves many objects and each state has many possible actions, search can be very expensive in this domain and manually coding the knowledge base is challenging. We gave the system 23 concepts and eight primitive skills, and we used two different goals that refer to configurations of one or two crates. For this study, we generated 96 random problems for use in the experiment. These problems include up to five crates, with difficulty levels ranging from one to 13-step solution paths depending on the initial configuration. The right-hand graph in Figure 2 shows the cumulative number of successes averaged over 20 different problem orderings for our algorithm.

The number of skills our system learns through this experiment is 42 on average, which is not sufficient to solve all the problems, but which does solve 70 percent of them. For this domain, although the hierarchical explanation-based algorithm generated some correct skills, it was very slow and could solve very few of the problems. As anticipated, because the learned skills are overspecific, it creates many skills (e.g., some 100 skills resulted from only four problems). However, we may be able to improve these results by generating multiple traces from a single problem. For example, if a goal G is associated with a skill sequence $S_1, S_2, S_3, \ldots, S_n$, the current technique learns one skill that achieves G using these subskills, but the subsequences $(S_2, S_3, \ldots, S_n), (S_3, S_4, \ldots, S_n), \ldots, (S_n)$ are also valid sequences for achieving G and can be used for training purposes. On the other hand, this would produce a more complex collection of skills that could slow response time further.

6. Related Research

Developing agents that can learn complex skills from experience has been a recurring goal in artificial intelligence. A common approach to this problem has focused on learning from delayed external rewards. Some methods (e.g., Moriarty et al., 1999) search through the space of the policies directly, whereas others (e.g., Kaelbling et al., 1996) estimate value functions for state-action pairs. In contrast, our approach differs by learning from traces of expert behavior rather than from exploration and by constructing hierarchical structures rather than flat policies.

There has been some work on learning control policies from expert traces. One of the main paradigms, known as behavioral cloning, transforms the observed traces into supervised training cases and induces reactive controllers that reproduce the behavior in similar situations. This approach typically casts learned knowledge as decision trees that determine which actions to take based on sensor input (e.g., Urbancic & Bratko, 1994). More recently, some efforts (e.g., Isaac & Sammut, 2003) have used goal information, stated as desired values for state variables, to improve the robustness of the learned controllers. In contrast, our approach infers intermediate structural goals by ex-

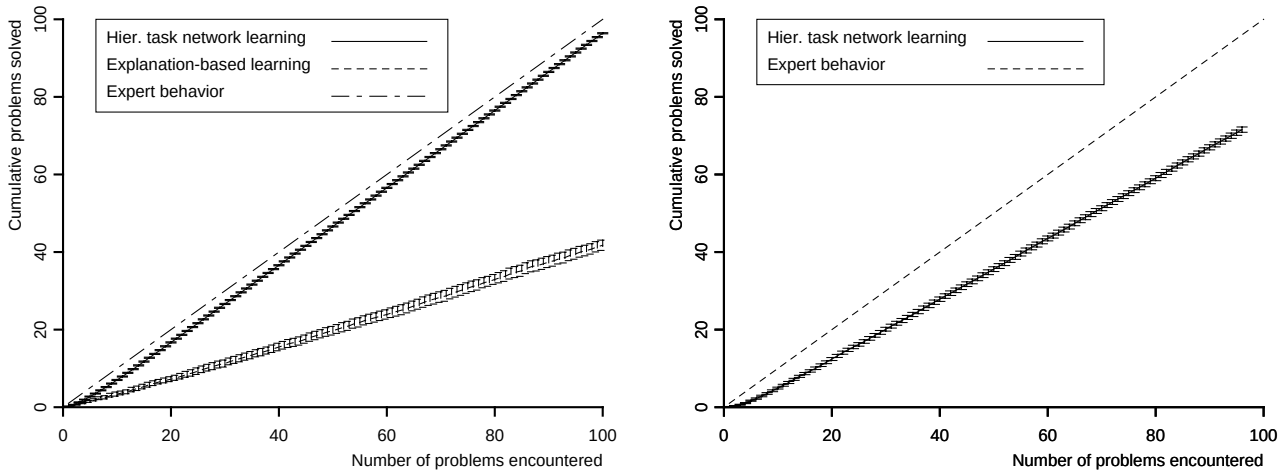

Figure 2. Cumulative number of solved problems in the Blocks World and Depots domains, with 95 percent confidence intervals. The diagonal lines depict the behavior of an expert who solves all the problems.

plaining the expert behavior in terms of the top-level goal and background knowledge. Moreover, our framework differs from most work on behavioral cloning by incorporating a hierarchical and relational representation for states and activities.

Könik and Laird (2004) report an approach to behavioral cloning that uses a relational representation and learns hierarchical controllers, but it requires the expert to annotate the trace with information about the start and end of activities at each level. A few other systems also learn hierarchical structures from expert behavior. Ilghami et al. (2005) describe a method for constructing hierarchical task networks, but they assume the hierarchical structure is given and use a version-space algorithm to determine conditions on methods. Tecuci's (1998) DISCIPLE acquires hierarchical rules, but it requires user information about how to decompose problems and the reasons for decisions.

Other research on learning skills by observing others' behavior has, like our own, utilized domain knowledge to interpret traces. Some work on explanation-based learning (e.g., Segre, 1987; Mooney 1990) took this approach, as did the paradigms of learning apprentices (e.g., Mitchell et al., 1985) and programming by demonstration (e.g., Cypher, 1993; Lau et al., 2003). Both often used analytic methods to generate candidate procedures, but neither focused on the acquisition of hierarchical skills, and programming by demonstration typically requires user feedback about candidate hypotheses. As noted earlier, our approach differs from explanation-based learning in that it retains the explanation structure and does not use deductive analysis to determine start conditions on new skills.

7. Concluding Remarks

In this paper, we have presented a new approach to learning hierarchical task networks from observation. We explained our reliance on expert traces to avoid the need for extensive search during problem solving. We also reviewed the notion of teleoreactive logic programs, a special class of task networks that are indexed by the goals they achieve, along with mechanisms for executing them to reach these objectives. After this, we described the details of our method for learning skill clauses from solution traces, which alternates between chaining off skills and concept definitions to generate an explanation of the expert's behavior. The system then creates one skill clause for each node in the explanation structure, using a simple technique for finding conditions on clauses that uses only local information.

Our approach incorporates ideas from a number of distinct traditions, including behavioral cloning, learning apprentices, and programming by demonstration, but it goes beyond these movements to construct hierarchical structures that generalize well to new problems. Moreover, experiments in two domains demonstrated that our method learns much more rapidly than explanation-based techniques, whether they construct flat macro-operators or take advantage of the learned skill hierarchy. This suggests that our approach to constructing hierarchical task networks offers advantages over more traditional techniques for learning plan knowledge.

Despite these encouraging results, our work on learning by observation remains in its early stages, and our current implementation makes some simplifying as-

sumptions that we should remedy in future work. One assumption is that the expert trace includes all relevant steps, whereas a more robust agent might infer occasional steps that it does not observe. Another is that learner knows the agent's goals, whereas an improved system would infer them from the trace and background knowledge. Our current method also assumes that the environment changes only when the agent takes some action, whereas future versions should handle settings in which some events are due to independent physical causes or other agents' behaviors.

In addition, our approach relies heavily on accurate knowledge about each skill's effects, whereas a more flexible learner would also include some ability to revise its action models based on observed results. Finally, the current system assumes that the expert has carried out the best sequence to achieve the goal, whereas a more sophisticated approach would analyze the trace for loops or other inefficiencies and remove them before learning. We hope to address each of these issues in future learning systems that combine domain knowledge with traces of expert behavior to acquire complex hierarchical task networks.

Acknowledgements

This paper reports research sponsored by DARPA under agreement FA8750-05-2-0283. The U. S. Government may reproduce and distribute reprints for Governmental purposes notwithstanding any copyrights. The authors' views and conclusions should not be interpreted as representing official policies or endorsements, expressed on implied, of DARPA or the Government. We thank Dongkyu Choi for discussions that contributed to the ideas presented in this paper.

References

Bacchus, F. (2001). AIPS'00 planning competition. *AI Magazine*, *22*, 47–56.

Cypher, A. (Ed.). (1993). *Watch what I do: Programming by demonstration*. Cambridge, MA: MIT Press.

Ellman, T. (1989). Explanation-based learning: A survey of programs and perspectives. *ACM Computing Surveys*, *21*, 163–221.

Ilghami, O., Nau, D. S., Muñoz-Avila, H., & Aha, D. W. (2002). CaMeL: Learning method preconditions for HTN planning. *Proceedings of the Sixth International Conference on AI Planning and Scheduling* (pp. 131–14). Toulouse, France.

Kaelbling, L. P., Littman, L. M., & Moore, A. W. (1996). Reinforcement learning: A survey. *Journal of Artificial Intelligence Research*, *4*, 237–285.

Könik, T., & Laird, J. (2004). Learning goal hierarchies from structured observations and expert annotations. *Proceedings of the Fourteenth International Conference on Inductive Logic Programming* (pp. 198–215). Porto, Portugal: Springer.

Langley, P., & Choi, D. (2006) Learning recursive control programs from problem solving. *Journal of Machine Learning Research*, *7*, 493–518.

Lau, T. A., Domingos, P., & Weld, D. S. (2003). Learning programs from traces using version space algebra. *Proceedings of the Second International Conference on Knowledge Capture* (pp. 36–43). Sanibel Island, FL: ACM Press.

Mitchell, T. M., Mahadevan, S., & Steinberg, L. I. (1985). LEAP: A learning apprentice for VLSI design. *Proceedings of the Ninth International Joint Conference on Artificial Intelligence* (pp. 573–580). Los Angeles, CA: Morgan Kaufmann.

Mooney, R. J. (1990). *A general explanation-based learning mechanism and its application to narrative understanding*. San Mateo, CA: Morgan Kaufmann.

Moriarty, D. E., Schultz, A. C., & Grefenstette, J. J. (1999). Evolutionary algorithms for reinforcement learning. *Journal of Artificial Intelligence Research*, *11*, 241–276.

Sammut, C. (1996). Automatic construction of reactive control systems using symbolic machine learning. *Knowledge Engineering Review*, *11*, 27–42.

Segre, A. (1987). A learning apprentice system for mechanical assembly. *Proceedings of the Third IEEE Conference on AI for Applications* (pp. 112–117).

Tecuci, G. (1998). *Building intelligent agents: An apprenticeship multistrategy learning theory, methodology, tool and case studies*. London: Academic Press.

Urbancic, T., & Bratko, I. (1994). Reconstructing human skill with machine learning. *Proceedings of the Eleventh European Conference on Artificial Intelligence* (pp. 498–502). Amsterdam: John Wiley.

Wilkins, D. E., & desJardins, M. (2001). A call for knowledge-based planning. *AI Magazine*, *22*, 99–115.

Zimmerman, T., & Kambhampati, S. (2003). Learning-assisted automated planning: Looking back, taking stock, going forward. *AI Magazine*, *24*, 73–96.