Learning Teleoreactive Logic Programs by Observation

Brian Salomaki, Dongkyu Choi, Negin Nejati, and **Pat Langley**
Computational Learning Laboratory
Center for the Study of Language and Information
Stanford University, Stanford, CA 94305 USA
{salomaki, dongkyuc, negin}@stanford.edu, langley@csli.stanford.edu

Abstract

In this paper, we review teleoreactive logic programs, a formalism for expressing goal-directed reactive controllers for physical agents. Although these programs previously had to be entered by hand or learned from problem solving, here we present a new way to acquire them from observation. The learning system observes the traces of primitive skills taken from an expert working on a known problem and learns new higher level skills based on this information. We explain the algorithm used for learning by observation and present initial results in two domains. Finally, we review related work and discuss directions for future research.

1. Introduction

Humans acquire knowledge in various ways. On their own, they learn how to achieve goal by problem solving and heuristic search. However, they can also acquire similar knowledge by observing others' behavior. This is extremely useful because one can absorb knowledge that other people have discovered through trial and error, without repeating their efforts. For physical activities, teaching often involves demonstrating a particular set of behaviors and letting the learner acquire new skills by observing the expert's actions. This ability is particularly important during early stages of learning, when people are acquiring many essential skills for the first time, building off of some basic knowledge about actions at the primitive level.

Our research on learning by observation builds on this intuition. We have developed an architecture (Choi et al., 2004) that supports hierarchical representations of skills, as well as methods for selecting and executing these skills in a physical environment. We have developed programs in this framework for tasks in the Blocks World, FreeCell solitaire, the Tower of Hanoi, and in-city driving. Although successful, we constructed these programs through careful hand crafting that required many man-hours to produce. In response, we augmented the system with the ability to solve novel problems using means-ends analysis and cache solutions in new skills (Choi & Langley, 2005). The extended architecture acquire a skill from each instance of an achieved goal, but it must sometimes carry out search through a large problem space to find this solution.

The focus of this paper is a new method for learning by observation that lets us "teach" the system by presenting it with a sequence of steps that accomplishes a given goal. Learning is based on analysis of this solution trace, thus eliminating the expensive search for a solution. Before describing this approach to learning, we review briefly the notion of teleoreactive logic programs and mechanisms for interpreting them. We then describe the new learning method, along with initial experimental results in two domains. In closing, we discuss related research in this area and propose some directions for additional work.

2. Teleoreactive Logic Programs

As noted above, our approach revolves around a representational formalism called teleoreactive logic programs, which incorporates ideas from logic programming, reactive control, and hierarchical task networks, and which is designed to support the execution and acquisition of complex procedures. A teleoreactive logic program consists of two knowledge bases. The first specifies a set of concepts that recognize classes of situations in the environment and describe them at higher levels of abstraction. A second knowledge base contains a hierarchical set of skills that the agent can execute in the world.

2.1. Representation of Knowledge

Concepts in our framework provide the agent with a hierarchical language to describe its beliefs about the environment. The concept definitions are stored in a conceptual long-term memory, with a syntax similar to Horn clauses. The lowest-level concepts (e.g., `on` in Table 1) are grounded in the agent's perceptual input from the environment, while other concepts (e.g., `clear` and `unstackable` in Table 1) build more complex ideas on top of these.

Skills describe the actions or subskills to execute under certain conditions. Each primitive skill clause has 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, each marked by an asterisk. More complex skills are specified by giving an ordered list of subskills in the place of executable actions. Taken together, skill definitions constitute a skill hierarchy, where each higher-level skill describes a reactive, partial plan composed of subskills. Table 2 shows some primitive skills for the Blocks World.

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)))
 :excludes ((clear ?blk2)))
(clear (?block)
 :percepts  ((block ?block))
 :negatives ((on ?other ?block))
 :excludes  ((on ?other ?block)))
(unstackable (?block ?from)
 :percepts  ((block ?block) (block ?from))
 :positives ((on ?block ?from)
             (clear ?block) (hand-empty)))
```

Table 2: Some 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)))
```

2.2. Inference and Execution

At the beginning of each cycle, the architecture receives updated contents for its perceptual buffer, which contains low-level sensory data on objects within the agent's field of perception. These perceptual elements are fed to an inference module that recognizes concepts based on their definitions. The module first instantiates primitive concepts at the lowest level by matching them against the perceptual elements and their numeric attributes. A concept instance consists of a predicate and objects that serve as its arguments. In a particular state, a given concept may match against more than one object or set of objects, resulting in multiple instances. After the module infers primitive concept instances from the percepts, it matches higher-level concepts against instances of the concepts that appear in their definitions. The entire concept hierarchy is instantiated in this bottom-up manner starting from low-level sensory inputs. Concept instances are not transferred across cycles; the inference process repeats on every cycle to ensure the system's beliefs are up to date.

High-level intentions, stored in a skill short-term memory, drive the agent's selection of skills to execute. On each cycle, the system finds all of the executable paths through the skill hierarchy that start from the highest-level intentions specified. Note that a skill path is not about a course of action over time; it describes the hierarchical context from the highest-level intention down to the corresponding low-level, directly executable actions. The applicability of each skill path depends on the applicability of every skill on that path. An individual skill is deemed applicable when all of its preconditions are satisfied based on environment information given by the entries in the perceptual buffer and the conceptual short-term memory. For reactivity, the system executes the first (leftmost) skill path that it finds to be applicable.

2.3. Problem Solving and Learning

Teleoreactive logic programs can be written manually, but the process is time consuming and prone to error. Therefore, the architecture includes a problem solver that chains primitive skills to solve novel tasks and a learning method that composes the solutions into executable programs. These mechanisms are interleaved with the execution process, and the problem solver is invoked whenever the agent encounters an impasse with no applicable skill paths. The system uses a variant of means-ends analysis (Newell et al., 1960), which chains backward from the given goal.

Each reasoning step can involve two different forms of chaining, depending on the type of knowledge used for the particular step. Skill chaining is used when the system has a skill that accomplishes the current subgoal, but the precondition of that skill is not met. The system first tries to achieve the precondition by executing another known skill or through problem solving, and then comes back to execute the original skill and thereby achieve the subgoal. Concept chaining occurs when the system does not have any skill clause that achieves the current goal directly. The system uses the concept definition of the current goal to decompose the problem into subgoals, and then tries to achieve all of the subgoals that are not already satisfied in the environment. The results at each step are stored in a goal stack.

When the system finds an applicable skill clause that will achieve the current subgoal, it executes the skill. Whenever a subgoal is achieved by such executions, the system immediately learns a new skill clause for it unless the new clause would be equivalent to an existing one. Once the information is stored, the system pops the subgoal and moves its attention to the parent. If the parent goal involved skill chaining, then the system executes the associated skill whose precondition has just become true, which in turn invokes learning and popping. If the parent goal involved concept chaining, one of the other unsatisfied subconcepts is pushed onto the goal stack or, if none remain, then the parent is popped. This process continues until the system achieves the top-level goal and learns all the applicable skill clauses from the particular solution path.

3. Learning by Observation

Although the architecture can learn using a single instance of problem solving, the process can be extremely slow for complicated goals. The agent must often perform an ex-

Table 3: Sample Blocks World input to the learning by observation method. The initial state for this problem has a three-block tower with C on B on A, and the goal is to clear block A. The numbers in the primitive skill instances are unique identifiers assigned by the execution architecture to each skill definition.

```
Goal:
  (CLEAR A)
Primitive Sequence:
  (((UNSTACK 1) C B) ((PUTDOWN 4) C T1)
   ((UNSTACK 1) B A))
Initial State:
  ((UNSTACKABLE C B) (HAND-EMPTY) (CLEAR C)
   (ONTABLE A T1) (ON C B) (ON B A))
```

pensive search through the problem space, as well execute the selected actions physically, which, in some domains like in-city driving, takes a significant time. In many cases, we would like to eliminate the search and speed learning by presenting the agent with a sequence of primitive skills that will achieve the goal.

3.1. Inputs to the Learning Method

When learning by observation, the agent is given the goal that the expert was working on, the sequence of primitive actions performed by the expert to achieve the goal, and the initial state of the environment. Table 3 shows an example of the input for a simple problem in the Blocks World domain.

The goal is given as a single concept instance, specifying both the concept and the objects to which it must apply. This assumption distinguishes our method from behavioral cloning (e.g., Sammut, 1996) and ensures that the system learns a useful skill that can be executed to accomplish similar goals in the future. The actions taken by the expert are provided as an ordered list of primitive skill instances. The actions should be limited to what is necessary to achieve the goal; there should be no additional actions taken after the goal is achieved.

Each state is represented as the complete set of concept instances that are true in the environment at a given time. From the given initial state, the learning agent can use the definitions of the skills in its long-term memory to determine the successive states.

3.2. A Method for Learning by Observation

To start the learning process, the agent uses the initial state of the environment, the sequence of primitives, and the definitions of the primitive skills to construct a sequence of states that parallels the sequence of primitives. Starting from the initial state, the agent considers the expert's first primitive action and reconstructs the new state of the environment after execution of that skill. It then starts from this new state and determines the third state by considering the effects of executing the second action, continuing in this manner until it has examined all of the primitive skill instances and constructed a complete sequence of states.

Once the state sequence has been determined, the agent works backward to learn the relevant skills, first looking at the overall goal and the final primitive skill executed. The agent chooses between two different learning methods, based on whether the final primitive skill directly achieves the goal as one of its effects or not. This choice echoes the choice between concept chaining and skill chaining in the means-ends problem solver described earlier.

If the current skill does not contain the goal as one of its effects, then it must achieve one of the goal's subgoals instead. 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 of the primitive actions taken between the state when the second-to-last subgoal became true and the final state (when the last subgoal became true) are assumed to contribute directly to achieving the final subgoal. Working from the state and primitive action sequences, the learning method constructs subsequences of the states and primitives that contribute to this subgoal. It then recursively calls the learning process, using the final subgoal as the overall goal and these subsequences as the complete primitive and state sequences.

The system repeats the same process, going all the way back through the sequences, finding the order in which the other subgoals became true and how they were achieved. After the primitive sequence is emptied and all of the recursive calls for learning on subgoals have returned, the agent learns a skill to achieve the overall goal by placing all of the subgoal-achieving skills in the order they occurred into the subskills field of the new skill, similar to concept chaining in the problem solver. Any subgoals already true in the initial state become the preconditions for executing this skill.

If instead current skill does directly contain the goal as one of its effects, then the agent learns a new skill with the goal concept as its head, the precondition of the final skill as its precondition, and a single subskill which is that final primitive.[1] The actual addition of skills to the agent's memory occurs through existing mechanisms, which already avoid adding duplicate skills.

After learning this skill, the system moves on to the next earlier action performed by the expert. The method now considers the precondition of the skill it has just processed and pops the top level of the state and primitive skill sequences, recursively calling the learning process as if the precondition had been the expert's original goal. Just as it assumes that the goal only becomes true in the final state, the learning algorithm assumes that the precondition only becomes true in the next-to-final state (now the final state in the new sequence). If the precondition had been true earlier, the expert should have executed the final goal-achieving skill as soon as it was possible.

If the level popped from the primitive sequence was the last remaining element and the list is now empty, the recursive learning process halts and returns to a higher level. In this case, the expert did nothing to achieve the current

[1] For the distinction between this learned skill and the primitive skill it is built on, see Choi and Langley (2005).

Table 4: Two of the higher-level skills in the Blocks World domain, as learned by observation.

```
UNSTACKABLE (?B ?A) id:  6
 :percepts ((BLOCK ?A) (BLOCK ?B))
 :start ((ON ?B ?A) (CLEAR ?B))
 :ordered ((HAND-EMPTY))
HAND-EMPTY NIL id:  5
 :percepts ((BLOCK ?B) (TABLE ?T1))
 :start ((PUTDOWNABLE ?B ?T1))
 :ordered ((PUTDOWN ?B ?T1))
```

precondition, so the learning method has no evidence from which it can learn a skill for achieving the precondition. On the other hand, when the new shorter sequence is not empty, the expert must have done some work to achieve the precondition of the current primitive skill. After making the recursive call and learning a new skill that achieves the precondition, the system adds another skill that achieves the current goal by composing the precondition-achieving skill with current primitive skill. This step is analogous to the one that results from successful skill chaining during problem solving.

4. Preliminary Results

At this stage, the learning by observation method has shown promising early results in the two domains on which we have tested it. The module already provides a viable alternative to programming teleoreactive logic programs by hand or using more expensive problem solving to learn skills, and it has even been able to provide results in a complex domain in which means-ends problem solving by itself does not produce consistent results.

4.1. Blocks World

In the Blocks World domain that we have used for examples throughout this paper, the learning method works as expected, acquiring complex skills that can be executed directly or used to facilitate further problem solving. To obtain training examples, we first ran the backward chaining problem solver on problems for which it can find suitable solutions, then saved the successful trace of the primitive skills it executed in achieving the goal. After starting fresh with only primitive skills and learning by observation from these traces, the agent acquires a set of skills identical to the ones learned during problem solving. Table 4 shows the first two skills acquired when learning by observation from the sample input shown in Table 3.

4.2. Depots

Depots is a more complicated domain that was introduced in the Third International Planning Competition. 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. Because a typical problem involves many objects and each state has many possible actions, this domain has proved challenging to code manually or to use with the existing problem solver to produce hierarchical skills.

However, when provided with a successful trace for a problem in the domain, the new learning system produces hierarchical skills that can then be executed directly to solve the same and similar problems. We have also observed the acquired skills make future problem solving faster and more likely to succeed. Our results to date are largely anecdotal, but we hope to carry out systematic experiments in this domain in the near future.

5. Related Research

Constructing 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. 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., Sammut, 1996; Urbancic & Bratko, 1994). Our work differs by utilizing information about the goal of the behavioral trace and, again, by creating hierarchical reactive skills rather than flat controllers. One counterexample comes from Könik and Laird (2004), who describe a behavioral cloning system that acquires hierarchical procedures. However, it relies on the expert to annotate the traces with the start and end of each skill, whereas our approach uses knowledge of the goal and primitive skills to infer the hierarchical structure without such assistance.

Other research on learning skills by observing others' behavior has also utilized domain knowledge to interpret the traces. Some work on explanation-based learning (e.g., Segre, 1987) took this approach, as did the paradigms of learning apprentices (e.g., Mitchell et al., 1985) and programming by demonstration (e.g., Cypher, 1993; Kaiser & Kreuziger, 1994). Both often used analytic methods to generate candidate procedures, but neither focused on the acquisition of hierarchical skills, and programming by demonstration typically require user feedback about candidate hypotheses. Tecuci's (1998) DISCIPLE system acquires hierarchical plan knowledge, but it requires user information about how to decompose problems and the reasons for the decision. Our approach differs from explanation-based learning (e.g., Ellman, 1989) in that it does not use deductive analysis to determine the start conditions on new skills.

6. Directions for Future Research

Although we have demonstrated promising preliminary results in the Blocks World and Depots domains, our work in this area has just begun. Future research should test our method for learning by observation in additional problems and domains to determine its robustness and reveal drawbacks. Obtaining experimental results of its capabilities for transferring learned knowledge to different problems, especially to ones more complex than the training tasks, will also be important.

Once we have established the learning method's basic functionality, additional research should integrate it into a more complete system for mixed-initiative problem solving. Combining observational learning with the architecture's existing methods for means-ends problem solving should let the system succeed at more complex tasks than either can handle alone. For example, when the system encounters a problem that it cannot solve on its own, it could request a solution trace from the human user. After learning the necessary skills from this trace, the system should be able to solve future problems that involve similar goals.

7. Concluding Remarks

In this paper, we have presented a new way to acquire teleoreactive logic programs from observation. After a brief review of such programs and the architecture that supports them, we explained our motivations for avoiding the search required by problem solving and proposed the idea of learning from an expert's action trace. We explained the details of the algorithm and promising demonstrations of its learning capabilities in two different domains, including evidence that it can utilize those learned structures on later tasks.

Teleoreactive logic programs are a variant of logic programs which support both goal-driven and reactive executions. The approach we have presented incorporates ideas from earlier work on behavioral cloning and learning by observation, but is novel in the sense that it combines goal information with the trace of actions to automatically construct hierarchical skills. Our work in this area is still in its early stages, but we hope to report more extensive results in the near future.

Acknowledgements

This research was funded by Grant HR0011-04-1-0008 from DARPA IPTO. Discussions with Tolga Könik, Seth Rogers, and Stephanie Sage contributed to the ideas presented here.

References

Choi, D., Kaufman, M., Langley, P., Nejati, N., & Shapiro, D. (2004). An architecture for persistent reactive behavior. *Proceedings of the Third International Joint Conference on Autonomous Agents and Multi Agent Systems* (pp. 988–995). New York: ACM Press.

Choi, D., & Langley, P. (2005). Learning teleoreactive logic programs from problem solving. *Proceedings of the Fifteenth International Conference on Inductive Logic Programming* (pp. 51–68). Bonn, Germany: Springer.

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*(2), 163–221.

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

Kaiser, M. & Kreuziger, J. (1994). Integration of symbolic and connectionist learning to ease robot programming and control. *ECAI-94 Workshop on Combining Symbolic and Connectionist Processing* (pp. 20–29). Amsterdam.

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.

Mitchell, T. M., Mahadevan, S., & Steinberg, L. (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.

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

Newell, A., Shaw, J. C., & Simon, H. A. (1960). Report on a general problem-solving program for a computer. *Information Processing: Proceedings of the International Conference on Information Processing* (pp. 256–264). UNESCO House, Paris.

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, England: 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.