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 first review the idea of teleoreactive logic programs. While teleoreactive logic programs previously had to be entered by hand or learned from problem solving, we present a new way of acquiring the programs without problem solving. The learning system observes the primitive skill traces of an expert working on a known problem, and learns new higher level skills based on this information. We explain in detail the algorithm used for learning by observation and provide initial results in two different domains. Finally, we review some related work and conclude with future directions of research.

1. Introduction

Humans acquire knowledge in various ways. On their own, they learn how to achieve a goal by trying different approaches and solving problems. They can also learn without solving problems, by observing others. 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, allowing the learners to 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 knowledge as well as selection and execution of skills in a physical environment. Programs coded using the architecture have solved problems in pole balancing, Blocks World, FreeCell solitaire, Tower of Hanoi, and in-city driving domains. While successful, these programs were developed through careful fabrication by humans and required many man-hours to produce. Therefore, we augmented the system with learning capabilities, and the ability to do problem solving to learn successful programs (Choi & Langley, 2005). The system can acquire knowledge from a single instance of problem solving for a particular goal, but sometimes must perform search through the problem space when given a nontrivial goal to achieve. The new ability to learn by observation allows us to "teach" the system by showing it a successful way to accomplish the goal and having it learn from that, thus eliminating the expensive search for a solution.

In this paper, we first briefly review the notion of teleoreactive logic programs and their interpretation. We then present the new learning method with initial experimental results. Finally, we discuss related work and conclude with future directions of research.

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 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 tests against 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 can be created by substituting an ordered list of subskills for the set 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 of the 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)))
```

2.2. Inference and Execution

At the beginning of each cycle, the system receives the updated contents of its perceptual buffer, which contains low-level sensory data on objects within the agent's field of perception. These perceptual elements initiate inference by invoking categorization based on concept definitions. At first, primitive concepts at the lowest level are instantiated directly from perceptual elements and their numeric attributes. An instance of a concept combines the head of the concept with the names of specific objects in the environment to which it applies. In a given state, a single concept may apply to more than one object or set of objects, resulting in multiple instances. After the primitive concepts are determined from the perceptual elements, higher-level concepts are inferred based on the instances of the primitive and other complex concepts by which they are defined. As a result, the whole hierarchy of concepts is instantiated through this bottom-up matching starting from low-level sensory inputs. Concept instances are not transferred across cycles; therefore, the matching process is repeated on every cycle, building from the updated perceptual buffer.

Agents have their highest-level intentions stored in the skill short-term memory. On each cycle, the agent 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 is dependent 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, our system has been augmented with a problem solver that chains primitive skills to solve novel tasks and a learning method that composes the solutions into executable programs. These mechanisms operate interleaved with the execution process, with the problem solver being invoked whenever the agent encounters an impasse with no applicable skill paths. Our system uses a variant of means-ends analysis (Newell et al., 1960), which chains backward from the given goal.

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)))
```

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 is used 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.

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))
```

3. Learning by Observation

Although the system can learn using a single instance of problem solving, the learning process can be extremely slow for complicated goals. The agent must often perform an expensive search through the problem space, as well as the physical execution of the selected actions, which, in some domains like in-city driving, takes a significant time. In certain cases, we want to eliminate the search and speed the learning process by presenting the agent with a known sequence of primitive skills that will achieve the goal.

3.1. Inputs to the Learning System

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.

A state is given as the complete set of concept instances that are true in the world 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. Learning Algorithm

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 learning 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.

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 learning agent 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 came true and the final state (when the last subgoal came true) are assumed to contribute directly to achieving the final subgoal. Pulling from the state and primitive action sequences, the learning agent 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 same process is repeated going all the way back through the sequences, finding the order in which the other subgoals came 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 that were already true in the initial state become the preconditions for executing this skill.

If, instead, the 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 identical skills.

After learning this skill, the agent moves on to the next earlier action performed by the expert. The agent now considers the precondition of the skill it has just processed and pops the top level off of the state and primitive skill sequences, recursively calling the learning process as if the precondition had been the expert's original goal. Like the assumption 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.

[1]For the distinction between this learned skill and the primitive skill it is built on, see Choi & 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))
```

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 precondition, so the agent learning by observation 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 learning agent adds an additional skill that achieves the current goal by composing the precondition-achieving skill with current primitive skill. This step is analogous to the skill chaining performed by the problem solver.

4. Preliminary Results

At this stage, the learning by observation method has shown promising early results in the two domains it has been tested on. It is already a viable alternative to programming teleoreactive logic programs by hand or using more expensive problem solving to learn relevant skills, and has even been able to provide results in a complex domain in which means-ends problem solving by itself cannot perform consistently.

4.1. Blocks World

In the Blocks World domain that has been used for examples throughout this paper, the learning by observation method works as expected, learning 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, and saved the successful trace of the primitive skills it executed in achieving the goal. After starting fresh with only primitive skills and just 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

The Depots domain is a more complicated domain introduced in the Third International Planning Competition in 2002. 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 of the high number of objects in a typical problem and the corresponding number of possible actions in each state, this domain has proven challenging to code for by hand or to use wth the existing problem solver to obtain executable skills.

Provided with a successful trace for a problem in the domain, however, the system can learn by observation and obtain skills that can then be executed directly to solve the same and similar problems, or used to make future problem solving faster and more successful.

5. Related Research

Building a physical agent with the ability to learn skills and control policies from its experiences has always been a need in the field of artificial intelligence. A common approach to this problem has been to focus 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. Unlike our new approach which addresses learning from observed behaviors of an expert, these methods are based on exploration and learning from trial and error.

There are a few frameworks which learn control policies from observed behaviors. One of the main streams of this research is known as behavioral cloning. The behavioral cloning framework observes traces of an expert's behavior, transforms them into supervised training cases and extracts operational models of the skills. This approach typically casts learned knowledge as decision trees that are used to decide which actions to take based on sensor input (e.g., Sammut, 1996; Urbancic & Bratko, 1994). In contrast to the main work in behavioral cloning, our system associates actions with a hierarchical structure of goals, learning skills that are robust and sufficiently general to apply in multiple contexts.

One similar approach that incorporates hierarchical knowledge of the goals that actions are working towards is presented by Könik and Laird (2004). While their system relies on the expert to annotate the observation traces with the goal that each step helps to achieve, ours requires only the overall goal to be given, and uses its knowledge of the concept and skill hierarchies to determine the current subgoals at each step.

Other work on learning skills by observing behaviors of an expert mostly rely on domain knowledge for interpreting the observed traces. Explanation-based learning (e.g., Segre, 1987), learning apprentices (e.g., Mitchell et al., 1985), and programming by demonstration (e.g., Cypher, 1993; Kaiser & Kreuziger, 1994) are examples of this category. In programming by demonstration, in addition to the demonstration trace itself, the expert needs to tell the system what relations have been maintained and/or achieved during the task. Tecuci's (1998) Disciple approach to building agents is similar in this regard, since the system must give the agent explanations as well as the examples. While the method we have introduced here is similar to explanation-based learning in that it relies on background knowledge provided to the system (Ellman, 1989), our system does not depend on deductive proof that each training case is an example of the target concept, and makes no associated guarantees.

6. Directions for Future Research

While our method already shows promising preliminary results in multiple domains, our work in this area has just begun. Future research should test learning by observation in additional problems and domains to ensure the accuracy and robustness of the method and implementation. Obtaining experimental results verifying the advantages of learning by observation will also be important.

With the basic functionality working well, future work should integrate the learning by observation method into a more complete system for mixed-initiative problem solving. Combining learning by observation with other methods of problem solving can allow the system to work more quickly and in more complex domains than could be handled by the other methods alone. When the system encounters a problem that it cannot solve on its own, it can request a working trace from the human expert guiding the system. After learning the necessary skills from this trace, the system should be able to solve future problems of similar complexity.

7. Concluding Remarks

In this paper, we have presented a new way of acquiring teleoreactive logic programs. After a brief review of such programs and the architecture that supports them, we explained our motivations for reducing search in problem solving and proposed the idea of learning by observation of an expert's action trace. We explained the details of the algorithm and promising early results of its learning capabilities in two different domains, along with its capacity to 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 presented here incorporates ideas from earlier work on behavioral cloning and learning by observation, but is unique in the sense that it uses hierarchical goal information in addition to the trace of actions involved to achieve it. Our work in this particular direction is still in the early stages, and we expect to report more results in near future.

Acknowledgements

This research was funded by Grant HR0011-04-1-0008 from DARPA IPTO.

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. *ECAI94 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.