Learning Teleoreactive Logic Programs from Problem Solving

Dongkyu Choi and Pat Langley

Computational Learning Laboratory
Center for the Study of Language and Information
Stanford University, Stanford, CA 94305 USA
dongkyuc@stanford.edu, langley@csli.stanford.edu

Abstract. In this paper, we focus on the problem of learning reactive skills for use by physical agents. We propose a new representation for such procedures, teleoreactive logic programs, along with an interpreter that utilizes them to achieve goals. After this, we describe a learning method that acquires these structures in a cumulative manner through problem solving. We report experiments in three domains that involve multiple levels of skilled behavior. We also review related work and discuss directions for future research.

1 Introduction

Humans typically acquire complex procedures in a cumulative manner, first mastering simple tasks and then utilizing them to learn higher-level abilities. At each stage, the learner must have reasonably automatized procedures before he can incorporate them into more sophisticated structures. The end result is a set of hierarchically organized skills that can be executed automatically, but at intermediate stages the person must rely at least partly on problem solving, which may require search to find solutions.

In this paper, we examine the task of learning such complex skills from a sequence of training problems. We are concerned with acquiring the structure and organization of skills, rather than tuning their parameters, which we view as a secondary learning issue. We assume the learner begins with primitive skills for the domain, including knowledge of their effects, and that training problems are presented in order of increasing complexity, much as in human instruction.

We focus here on procedures that involve action in the world, but ones that are more complex than those usually studied in research on reinforcement learning (Sutton & Barton, 1998) and behavioral cloning (Sammut, 1996). We assume that the agent encodes its knowledge in a formalism – teleoreactive logic programs – designed specifically for such tasks, and our learning methods take advantage of this notation to constrain the acquisition process. As we will see, these are similar in spirit to early techniques for learning macro-operators and search-control rules, but they also differ in important ways.

In the next section, we specify the formalism used to encode initial and learned knowledge, along with performance mechanisms that interpret them to

Table 1. Examples of concepts from an in-city driving domain.

```
(parked (?self ?lane)
 :percepts  ((self ?self speed ?speed))
 :positives ((in-rightmost-lane ?self ?lane)
             (stopped ?self)))

(in-lane (?self ?lane)
 :percepts ((self ?self segment ?sg)
            (lane-line ?lane segment ?sg dist ?dist))
 :tests    ((> ?dist -10)
            (<= ?dist 0)))
```

produce behavior. After this, we present an approach to problem solving on novel tasks and a learning mechanism that transforms the results of this process into executable logic programs. Next, we report experimental studies of the method in three domains, including an in-city driving task that we use to illustrate our ideas. In closing, we review related work on learning and consider directions for additional research.

2 Teleoreactive Logic Programs

As noted, our approach revolves around a representational formalism, called teleoreactive logic programs, that are designed to support the execution and acquisition of complex procedures. We refer to these structures as "logic programs" because their syntax is similar to the Horn clauses used in Prolog and related languages. We have borrowed the term "teleoreactive" from Nilsson (1994), who used it to refer to systems that are goal driven but that also react to their current environment. His examples incorporated symbolic control rules but were not cast as logic programs, as we assume here.

A teleoreactive logic program consists of two knowledge bases. One specifies a set of concepts that recognize classes of situations in the environment and describe them at higher levels of abstraction. These monotonic inference rules have the same semantics as traditional Horn clauses and a similar syntax. Each clause includes a single head, stated as a predicate with zero or more arguments, along with a body that includes one or more positive literals, negative literals, or arithmetic tests. The same head can appear in more than one clause, expressing different ways to satisfy the named concept.

We distinguish between primitive clauses, which refer only to percepts that the agent can perceive in the environment, and complex conceptual clauses, which refer to other concepts in their body. Specific percepts play the same role as ground literals in traditional logic programs, but, because they can change rapidly, we do not consider them part of the program. Table 1 presents some concepts from an in-city driving domain. The concept *parked* is defined in terms of the concepts *in-rightmost-lane* and *stopped*, whereas *in-lane* is defined in terms of the percepts *self* and *lane-line*, along with arithmetic tests on their attributes.

Table 2. Examples of skills from an in-city driving domain. The complex skill (first) has typed variables, a start condition, and a set of ordered subskills. The primitive (second) skill has a set of actions (marked by an asterisk) and effects instead of subskills.

```
(driving-in-segment (?self ?sg ?lane)
 :percepts ((lane-line ?lane) (segment ?sg) (self ?self))
 :start    ((steering-wheel-straight ?self))
 :skills   ((in-lane ?self ?lane)
            (centered-in-lane ?self ?sg ?lane)
            (aligned-with-lane-in-segment ?self ?sg ?lane)
            (steering-wheel-straight ?self))
(steering-wheel-straight (?self)
 :percepts ((self ?self))
 :start    ((steering-wheel-not-straight ?self))
 :actions  ((*straighten))
 :effects  ((steering-wheel-straight ?self)))
```

A second knowledge base contains a set of skills that the agent can execute in the world. Each skill clause includes a single head (a predicate with zero or more arguments) and a body that specifies a single start condition and one or more components. Primitive clauses refer to executable actions that affect the environment. They also specify the effects of their execution, stated as literals that hold after their completion, and may state requirements that must hold during their execution. Primitive skill clauses are similar in structure and spirit to STRIPS operators, although they may be executed in a durative manner.

In contrast, complex skill clauses specify how to decompose activity into subskills. Because a skill may refer to itself, either directly or through a subskill, the formalism supports recursive definitions. For this reason, nonprimitive skills do not specify effects, which can differ for different levels of recursion, nor do they state requirements. However, the head of each complex skill corresponds to some concept that the skill aims to achieve, with its head using the same predicate and taking the same number of arguments as the concept. This connection between skills and concepts figures centrally in the learning methods we describe later. Table 2 presents some skills for the driving domain, including a complex skill, *driving-in-segment*, and a component primitive skill, *steering-wheel-straight*.

Note that every skill S can be expanded into one or more sequence of primitive skills. For each skill S in a teleoreactive logic program, if S has concept C as its head, then every expansion of S into such a sequence must, if executed successfully, produce a state in which C holds. This second constraint does not guarantee that, once initiated, the sequence will achieve C, since other events may intervene or it may encounter states in which one of the primitive skills may not apply. However, if the sequence of primitive skills can be run to completion, then it will achieve the goal literal C. The approach to learning that we report later is designed to acquire programs with this characteristic, although we do not yet have a formal proof to that effect.

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 we have described elsewhere (Choi et al., 2004), the basic architecture proceeds in discrete cycles, in each case invoking an inference process that elaborates on the agent's perceived state and an execution process that generates behavior in the environment.

The inference module operates in a bottom-up, data-driven manner that starts from descriptions of perceived objects, such as (segment G1113 street 1 dist -5.0 latdist 15.0 dir WE), and deduces all beliefs that they imply in combination with the conceptual clauses, such as (in-lane ME G1213). This inference process augments the agent's perceptions with higher-level descriptions of the environment that may be useful for its decision making. Although this mechanism reasons over structures similar to Horn clauses, its operation is closer in spirit to the bottom-up elaboration process in Soar (Laird et al., 1986) than to the query-driven reasoning in Prolog.

In contrast, the execution module proceeds in a top-down manner, starting from high-level intentions, such as (delivered-package ME package5), and finding applicable paths through the skill hierarchy that terminate in primitive skills with executable actions, such as ($*$steer -0.5). A *skill path* is a chain of skill instances that starts from the agent's top-level intention and descends the skill hierarchy, unifying the arguments of each subskill consistently with those of its parent. A path is *applicable* if the concept instance that corresponds to the intention is not satisfied, if the requirements of the terminal (primitive) skill instance are satisfied, and if, for each skill instance in the path not executed on the previous cycle, the start conditions are satisfied. This last constraint is necessary because skills may take many cycles to achieve their desired effects, making it important to distinguish between their initiation and their continuation.

Both conceptual inference and skill execution play essential roles in complex domains like in-city driving. On each cycle, the agent perceives nearby objects and infers instances of conceptual relations that they satisfy. For each intention, the system then uses these beliefs to check the conditions on skill instances and to determine which paths are applicable, which in turn constrains which actions it executes. The environment changes, either in response to these actions or on its own, and the agent begins another inference-execution cycle. This looping continues until the concepts associated with each of the agent's top-level intentions are satisfied, when it halts.

The interpreter incorporates two preferences that provide a balance between reactivity and persistence. First, given a choice between two or more subskills, it selects the first one for which the corresponding concept instance is not satisfied. This bias supports reactive control, since the agent reconsiders previously completed subskills and, if unexpected events have undone their effects, reexecutes them to correct the situation. Second, given a choice between two or more applicable skill paths, it selects the one that overlaps most with the path executed on the previous cycle. This bias encourages the agent to keep executing a high-level skill it has started until it achieves the associated goal or becomes inapplicable.

4 Problem Solving and Learning Mechanisms

Although one can construct teleoreactive logic programs manually, this process is time consuming and prone to error. In response, we have developed a problem solver that chains primitive skills to solve novel tasks and an associated learning method that composes the solutions into executable programs, which we describe in this section. Both mechanisms are interleaved with the execution process, with the problem solver being invoked whenever the agent encounters a situation for which it finds no applicable skill paths. As in Laird et al.'s Soar, problem solving and learning are driven by impasses, although the details are quite different.

4.1 Means-Ends Problem Solving

As noted, our system resorts to problem solving when there are no applicable skill paths that would take it toward the current goal. We utilize a variant of means-ends analysis (Newell et al., 1960) which chains backward from the goal, pushing the result of each reasoning step onto a goal stack that stores information about the agent's efforts toward achieving the goal. As the pseudocode in Table 3 indicates, two distinct forms of chaining play a role in problem solving.

Backward chaining off a skill involves retrieving a skill clause with effects or a head that indicates its execution would achieve the current goal. If such a clause exists in skill memory, the system associates an instance of this clause with the goal.[1] If the clause's start condition is met, the system executes the clause instance in the environment until it achieves the goal, which is then popped from the stack. If the condition is not satisfied, the system makes it the current goal by pushing it onto the stack.

If the problem solver cannot find any skill clause that would achieve the current goal, it resorts to concept chaining. Here it uses the definition of the goal concept to decompose the problem into subgoals. Some subgoals may already be satisfied in the current situation, which the system stores as such with the current goal. If more than one subgoal is unsatisfied, the problem solver selects one at random and makes it the current goal by pushing it onto the goal stack.

The system continues along these lines, pushing new goals onto the stack until it finds one it can achieve with an applicable skill clause. In such cases, it executes the skill and pops the goal from the stack. If the parent goal involved skill chaining, then this leads to execution of its associated skill and achievement of the parent, which is in turn popped. 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.

Of course, the problem-solving procedure must make decisions about which skills to select during skill chaining and the order in which it should tackle

[1] When there are multiple relevant clauses, the problem solver selects the one with the fewest conditions unsatisfied in the current situation. Because skills always have a single start condition, this means expanding the concept into its primitive components. If the candidates tie on this criterion, then it selects the clause that requires fewer expected steps, and if ties occur on this dimension, it selects one at random.

Table 3. Pseudocode for means-ends problem solving and associated learning through goal-driven composition of component skills.

```
Solve(G)
  Push the goal literal G onto the empty goal stack GS.
  On each cycle,
     If the top goal G of the goal stack GS is satisfied,
     Then pop GS and let New be Learn(G).
          If G's parent P involved skill chaining,
          Then store New as P's first subskill.
          Else if G's parent P involved concept chaining,
               Then store New as P's next subskill.
     Else if the goal stack GS does not exceed the depth limit,
          Let S be the skill instances whose heads unify with G.
          If any applicable skill paths start from an instance in S,
          Then select one of these paths and execute it.
          Else let M be the set of primitive skill instances that
                  have not already failed in which G is an effect.
               If the set M is nonempty,
               Then select a skill instance Q from M.
                    Store Q with goal G as its last subskill.
                    Push the start condition C of Q onto goal stack GS.
                    Mark goal G as involving skill chaining.
               Else if G is a complex concept with the unsatisfied
                       subconcepts H and with satisfied subconcepts F,
                    Then if there is a subconcept I in H that has not yet failed,
                         Then push I onto the goal stack GS.
                              Store F with G as its initially true subconcepts.
                              Mark goal G as involving concept chaining.
                         Else pop G from the goal stack GS.
                              Store information about failure with G's parent.
                    Else pop G from the goal stack GS.
                         Store information about failure with G's parent.

Learn(G)
  If the goal G involves skill chaining,
  Then let $S_1$ and $S_2$ be G's first and second subskills.
       If subskill $S_1$ is empty,
       Then return the literal for clause $S_2$.
       Else create a new skill clause N with head G,
            with $S_1$ and $S_2$ as ordered subskills, and
            with the same start condition as subskill $S_1$.
            Return the literal for skill clause N.
  Else if the goal G involves concept chaining,
       Then let $C_{k+1}$, ..., $C_n$ be G's initially satisfied subconcepts.
            Let $C_1$, ..., $C_k$ be G's stored subskills.
            Create a new skill clause N with head G,
            with $C_{k+1}$, ..., $C_n$ as ordered subskills, and
            with the conjunction of $C_1$, ..., $C_k$ as start condition.
            Return the literal for skill clause N.
```

subconcepts during concept chaining. The system may well make the incorrect choice at any point, which leads it to pop the current goal and backtrack when the goal goal reaches its maximum depth or when it has no alternatives it has not already tried. As a result, it carries out depth-first search through the problem space, which can require considerable time on some tasks.

Figure 1 shows an example of the problem solver's behavior in our in-city driving domain. When the system is given the objective *driving-in-segment*, it looks for any executable skill with this goal as its head. When this fails, it looks for a skill that has the objective as one of its effects. Since it has no such skills, it chains off the concept definition, which involves four subconcepts. The first

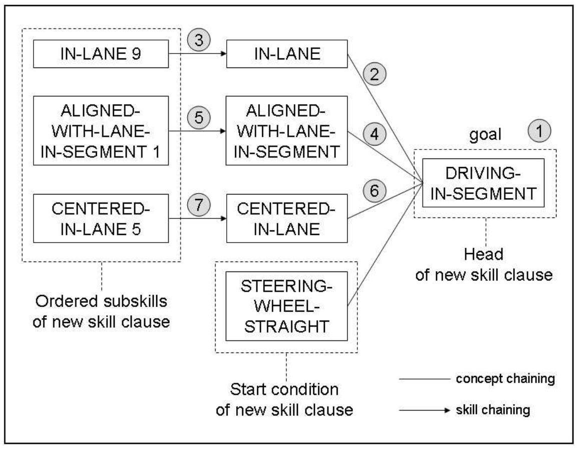

Fig. 1. A trace of successful problem solving in the in-city driving domain. Circled numbers correspond to the steps explained in the text.

three are not satisfied, from which the system chooses *in-lane* as its subgoal. It finds skill clauses with this head and selects (in-lane 9), which is applicable and which it executes to achieve the subgoal.

Later, the problem solver does the same for the subconcepts *aligned-with-lane-in-segment* and *centered-in-lane*, selecting the skills (aligned-with-lane-in-segment 1) and (centered-in-lane 5), which the system executes in turn to achieve them. At this point, it notes that the original goal is satisfied. This example describes a trace of successful problem solving, but we have omitted missteps that require backtracking and search for the sake of clarity.

4.2 Learning through Goal-Driven Composition

Fortunately, learning can transform the results of search into a teleoreactive logic program that can be executed efficiently. Whenever the agent achieves a goal during problem solving, it stores a new skill clause, unless its solution involves immediate execution of an existing clause or unless the new clause would be equivalent to an existing one. The learning mechanism, which we call *goal-driven composition*, operates somewhat differently for each form of chaining.

When the agent reaches an objective through skill chaining, say by achieving a goal G by first applying skill S_1 to satisfy the start condition for S_2 and executing the skill S_2, the learning mechanism constructs a new clause[2] with head G and ordered subskills S_1 and S_2. The start condition for the new clause is the same as that for S_1, since when S_1 is applicable, the successful completion of this skill will ensure the start condition for S_2, which in turn will achieve G. This differs from traditional methods for constructing macro-operators, which

[2] If the skill S_2 can be executed without invoking another skill to meet its start condition, the method creates a new clause G with S_2 as its only subskill.

analytically combine the preconditions of the first operator and those preconditions of later operators it does not achieve. However, S_1 was either selected because it achieves S_2's start condition or it was learned during its achievement, both of which mean that S_1's start condition is sufficient for the composed skill.

In contrast, successful concept chaining leads to the creation of slightly different skill clauses. Suppose the agent achieves a goal concept G by satisfying the subconcepts $G_1, \ldots, G_k$, in that order, while subconcepts $G_{k+1}, \ldots, G_n$ were true at the outset. In this case, the system constructs a new skill clause with head G and the ordered subskills $G_1, \ldots, G_k$.[3] In this case, the start condition for the new clause is the conjunction of subgoals that were already satisfied beforehand. This prevents execution of the learned clause when some of $G_{k+1}, \ldots, G_n$ are not satisfied, in which case the sequence $G_1, \ldots, G_k$ may not achieve the goal G.

Goal-driven composition operates in a bottom-up fashion, with new skills being formed whenever a goal on the stack is achieved. The method is fully incremental, in that it learns from single training cases, and it is interleaved with problem solving and execution. Unlike most techniques for learning macro-operators, it can acquire both disjunctive and recursive skills. Moreover, learning is cumulative in that skill clauses learned from one problem are available for use on later tasks. However, the system invokes a learned clause only when it is applicable in the current situation, so the problem solver never chains off its start condition. Mooney (1989) relied on a similar strategy to avoid the utility problem (Minton, 1990), in which learned knowledge leads to slower behavior.

Our approach to learning takes advantage of three insights that make it effective. First, although means-ends analysis is seldom used in the AI planning community, it has the distinct advantage that, when successful, it produces an AND tree that decomposes the original problem into subproblems. This determines the structure of the learned clauses. Second, the problem-solving trace indicates the goal literal being pursued in each subproblem, which provides the head for the learned clause. Because the same goal may be achieved in different ways, this leads naturally to both disjunctive and recursive structures. Finally, the skill clauses in a teleoreactive logic program are interpreted not in isolation but as parts of chains through the skill hierarchy. This lets the learning method store very abstract conditions with new clauses without a danger of overgeneralization. Taken together, these features make goal-driven composition a simple yet powerful approach to learning logic programs for reactive control.

5 Experimental Studies of Learning

Preliminary studies in a number of domains suggested that the learning mechanism described above constructs appropriate teleoreactive logic programs. Given the same or isomorphic problems, the agent retrieves and executes the learned programs in a reactive manner, without resorting to means-ends problem solving. These informal results encouraged us to test the system in a more dynamic

[3] Each of these subskills was either already known and used to achieve the associated subgoal or it was learned from the successful solution of one of the subproblems.

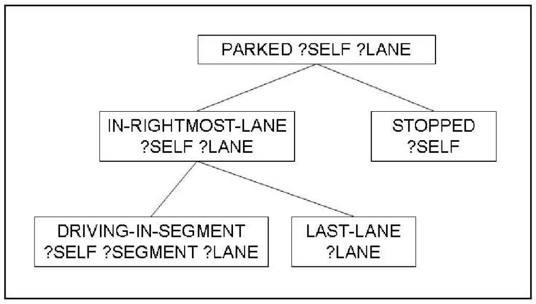
PARKED ?SELF ?LANE
IN-RIGHTMOST-LANE ?SELF ?LANE
STOPPED ?SELF
DRIVING-IN-SEGMENT ?SELF ?SEGMENT ?LANE
LAST-LANE ?LANE

Fig. 2. Portion of the concept hierarchy given to the system that it uses to decompose the goal instance (parked ME LANE1). Skills that achieve these concepts are learned through problem solving.

domain that involved in-city driving, which we present below. We also carried out more systematic experiments with two domains that are less dynamic but that involve recursive structure, which should let the learned programs scale to more complex problems.

5.1 In-City Driving

In-city driving is a demanding task that involves the reactive use of complex skills. To study this problem, we have implemented a simulated environment that makes simplifications but retains the dynamic nature and complexity of the real world. Objects in the environment are represented as rectangles of various sizes on a Euclidean plane. These include static objects like road segments, intersections, lane lines, and buildings, but they also include moving vehicles.

One vehicle is controlled by the agent, whereas others follow standard driving customs but make random decisions about whether to drive through intersections or turn. The agent can invoke actions for accelerating, decelerating, and turning its steering wheel left or right. These inputs affect the associated control variables of the vehicle according to realistic physical laws. The agent can perceive objects around it up to 60 feet away, including other vehicles and buildings, each of which is described in agent-centered polar coordinates that give its distance, angle, relative velocity, and angular velocity. The agent also perceives its own properties, including its current speed and its steering wheel angle.

For this domain, we provided the system with 19 concepts and eight primitive skills to achieve a goal that we stated as an instance of some defined concept. The particular task we report in this paper involves achieving the goal (parked ME LANE1), which has subconcept instances (in-rightmost-lane ME LANE1) and (stopped ME). The first subconcept can be decomposed further into (driving-in-segment ME SEGMENT1 LANE1) and (last-lane LANE1), as shown in Figure 2. As the concepts clarify, this task can be done by first changing to the rightmost lane (by achieving *driving-in-segment* and *in-rightmost-lane*, in that order), if the vehicle is not already in that lane, and then slowing to a stop.

Table 4. Skill clauses learned from a run in the in-city driving domain.

```
driving-in-segment (?ME ?G1101 ?G1152)
 :percepts ((lane-line ?G1152) (segment ?G1101) (self ?ME))
 :start    ((steering-wheel-straight ?ME))
 :skills   ((in-lane ?ME ?G1152)
            (centered-in-lane ?ME ?G1101 ?G1152)
            (aligned-with-lane-in-segment ?ME ?G1101 ?G1152)
            (steering-wheel-straight ?ME))
in-rightmost-lane (?ME ?G1152)
 :percepts ((self ?ME) (lane-line ?G1152))
 :start    ((last-lane ?G1152))
 :skills   ((driving-in-segment ?ME ?G1101 ?G1152))
parked (?ME ?G1152)
 :percepts ((lane-line ?G1152) (self ?ME))
 :start    ( )
 :skills   ((in-rightmost-lane ?ME ?G1152)
            (stopped ?ME))
```

Table 4 shows the teleoreactive logic program acquired from one learning run. All the skill clauses are constructed from concept chaining. During problem solving, the objective (driving-in-segment ME G1101 G1152) of the first skill clause is achieved by *in-lane*, *centered-in-lane*, *aligned-with-lane-in-segment*, and *steering-wheel-straight*, in that order. When execution of the first subskill was started, the concept (steering-wheel-straight ME) was true, so it is included in the start condition. Since the goal (in-rightmost-lane ME G1152) was at the next level of the stack, the system creates the second skill clause, *in-rightmost-lane*, immediately after the first clause and used the first learned clause *driving-in-segment* as its only subskill. Again, because the concept instance (last-lane G1152) held when the chaining that led to this skill clause began, it is included in the start condition. In turn, it uses the skill *in-rightmost-lane* as the first subskill of the skill clause *parked* followed by the primitive clause *stopped.* Note that these learned clauses are organized hierarchically, but they all expand into primitive skills with executable actions. When combined with the original skills, the learned program shows the desired behavior. Moreover, they generalize correctly to situations with different numbers of lanes and other starting lanes.

These results are encouraging, but we also desired to answer two questions that required a more formal experiment. First, we wanted to know whether the learning method produces teleoreactive logic programs that are more effective for the task than the primitive skills combined with means-ends problem solving. To this end, we ran the system on the task of achieving (parked ME LANE1) both with learning turned on and with it turned off. Second, we wanted to determine whether cumulative learning on subtasks of increasing complexity produced more rapid improvement than learning on the goal task. For the former, we first let the system master the component task of achieving (driving-in-segment ME

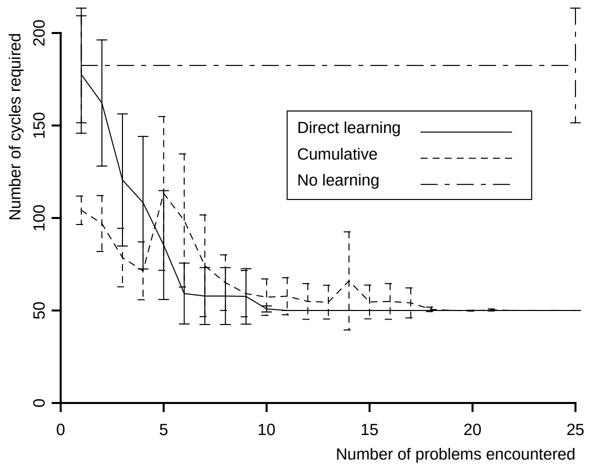

Fig. 3. Execution cycles required to achieve the goal (parked ME) in the driving domain as a function of the number of trials with no learning, non-cumulative learning, and cumulative learning. Each learning curve shows the mean over 25 runs and 95 percent confidence intervals.

SEGMENT1 LANE1) and then (in-rightmost-lane ME LANE1) before finally letting it learn how to achieve the top-level goal, (parked ME LANE1).

In each of these three conditions, we ran the system 25 times on the driving task. We measured the number of decision-making cycles the system took to solve the task on each trial, which differ due to the randomness of selection in the chaining process. Based on these data, we computed the mean and 95% confidence interval as a function of the trial number, which we plot in Figure 3. With learning disabled, performance on this task does not improve and the agent continues to take 180 cycles to solve the problem. With learning activated, the number of cycles dramatically decreases over the first few trials and converges to about 50 cycles after ten trials, which is the fewest cycles needed for this particular task. This suggests that learning is effective for the problems that arise in the in-city driving domain.

We also compared the basic learning condition to the cumulative learning case, in which we presented the system with subgoals in order of increasing difficulty. The system begins taking fewer cycles than the non-cumulative case, mainly because we gave it a relatively simple goal first. Around the fifth trial, it completes learning on the first simple problem and moves on to the second one. Since this task is new to the system, it needs more cycles for problem solving, which produced the peak on the graph. However, this peak is still lower than the level at which non-cumulative learning began, since the learned skills from the first task reduced effort. After learning to achieve the second subgoal, the most difficult task became noticeably easier than without prior learning, and the graph shows no detectable peak. Even though cumulative learning requires more trials to complete the learning process, on average it needs fewer cycles per trial.

Table 5. Recursive skills learned from a Blocks World problem in which C is on B and B is on A in the initial state and the goal is to make A clear. Some start conditions have been expanded for the sake of clarity.

```
unstackable (?C ?B)                hand-empty ( )
:percepts  ((block ?B)(block ?C))  :percepts  ((block ?D)(table ?T1))
:start     ((on ?C ?B)(hand-empty)) :start     ((putdownable ?D ?T1))
:skills    ((clear ?C)(hand-empty)) :skills    ((putdown ?D ?T1))

clear (?B)                         holding (?D)
:percepts  ((block ?C)(block ?B))  :percepts  ((block ?D)(block ?C))
:start     ((on ?C ?B)(hand-empty)) :start     ((unstackable ?D ?C))
:skills    ((unstackable ?C ?B)    :skills    ((unstack ?D ?C))
            (unstack ?C ?B))

clear (?C)
:percepts  ((block ?D)(block ?C))
:start     ((unstackable ?D ?C))
:skills    ((unstack ?D ?C))
```

Although in-city driving is a challenging physical domain, its structure does not take full advantage of our method's capabilities. We predict that, combined with a cumulative training regime, it will be especially effective in domains with recursive structure, since its ability to learn recursive logic programs will let it train on simple problems and generalize to more complex ones. To test this prediction, we also carried out experiments with two other domains, the Blocks World and FreeCell solitaire, that are known to have recursive structure,

5.2 Blocks World

The Blocks World consists of a table with cubical blocks and a gripper. For this domain, we provided the system with nine concepts and four primitive skills, along with one concept for each of four distinct goals. These are sufficient, in principle, to solve all the problems in the domain, but means-ends analysis would require extensive search when there are more than a few blocks. Instead, we want a teleoreactive logic program that can solve problems with arbitrary numbers of blocks without significant search. For this study, we developed a simulated environment that let the agent perceive the positions of objects and manipulate blocks by grasping, lifting, carrying, and ungrasping them. Table 5 presents the recursive skills learned from one training problem that required clearing the lowest object in a stack of four blocks.

To determine whether the control programs our method constructs are useful on more complex tasks, we carried out a transfer experiment that involved two conditions. In one condition, we presented the system with 20 training problems with three blocks, another 20 problems with four blocks, and a third set of the same size with five blocks. Each problem involved one of four conjunctive

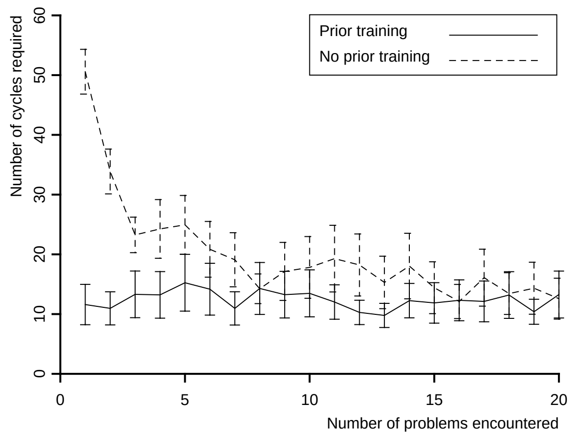

Fig. 4. Execution cycles required to solve a 20-block problem in the Blocks World as a function of the number of training tasks with and without prior training on simpler problems. Each learning curve shows the mean over 500 different training orders and 95 percent confidence intervals.

goals that referred to configurations of one, two, or three blocks. After this, we presented the system with 20 new problems that referred to analogous goals but that each involved 20 blocks. In the second condition, we asked the system to solve the same 20 tasks, but without the benefit of working on the simpler problems. Learning was active in both conditions, but the former had the benefit of prior training when it encountered the transfer set.

Figure 4 shows the number of execution cycles required for both conditions, averaged over 500 runs using transfer problems with different randomized orders. For each problem, we let the system run a maximum of 50 cycles before starting over and attempt the task at most ten times before giving up. In both conditions, the system managed to solve 99 percent of the problems, but there was considerable difference in the effort required. When the system worked on simpler problems first, it constructed a recursive logic program that, in nearly all cases, handled the 20 block tasks without resorting to means-ends analysis. This condition shows no improvement because the system had learned all there was to learn in the simpler setting. However, this does not mean they would not challenge traditional learning methods that cannot acquire recursive structures.

In contrast, without the benefit of this prior experience, the system had to invoke its problem solver, leading to much longer runs on the initial tasks. Even in this case, the learning mechanism rapidly acquired a teleoreactive logic program, with the system reaching apparent asymptotic performance after only five problems. However, its performance under this condition did not quite reach the same level as under the transfer condition, suggesting that training on simpler problems provides an overall advantage. We repeated this study with tasks that involved 30 blocks and obtained almost identical results.

5.3 FreeCell Solitaire

FreeCell is a solitaire game that involves stacks of cards on eight columns, all faced up and visible to the player. There are four free cells, which serve as temporary holding spots for a single card at a time, and four foundation cells that correspond to four different suits. The goal is to move all the cards on the eight columns to the foundation cells in ascending order and grouped by suit. At any given time, only the cards on the top of the stack on each column and the ones in free cells are available for movement, and they can shift to a free cell, to the proper foundation column, or to an empty column. We again provided a simulated environment that let the agent make legal moves, as well as perceive card locations and the status of cells.

For this domain, we provided the system with 24 concepts and 12 primitive skills that are sufficient to handle any initial configurations capable of solution. But again, means-ends analysis may require an inordinate amount of effort to handle problems with more than a few cards or convenient configurations. We hoped our learning mechanism would acquire a teleoreactive logic program that could solve arbitrary FreeCell tasks in the simulated environment with little or no need to invoke the problem-solving module. To this end, we carried out a transfer experiment similar to that we reported for the Blocks World.

In the transfer condition, we trained the system on 20 randomly generated FreeCell tasks that involved eight cards, another 20 problems with 12 cards, and a third set of the same size with 16 cards. We then asked the system to tackle a set of 20 harder tasks that involved configurations of 20 cards. In the control condition, we presented the system with the similar 20 card problems but we did not let it work on the simpler ones first. Figure 5 displays the number of execution cycles needed for each condition, averaged over 300 random sequences of the harder problems. As for the Blocks World, the system in the transfer condition shows no improvement beyond what it gained from working on the simpler tasks. However, in the control condition, the system requires much more effort at the outset and it continues to require substantially more effort than the version that trained on easier problems.

In this domain, the system cannot solve all of the more complex problems. The recursive program learned from simpler problems, in the transfer condition, handles around 72 percent of the 20 card FreeCell tasks. Prior experience leads to the creation of useful structures, but the training problems do not produce any skills for moving from one column to another, which are needed for some 20 card tasks. However, without the benefit of this earlier training, the system can initially solve only 39 percent of these problems, and its solution probability remains below that for the other condition through the learning curve.[4] When we used harder problems for the transfer set, this effect was far more pronounced. This result provides even stronger evidence that our method benefits from a cumulative approach to learning.

[4] With learning turned off, the system could solve none of the 20 card problems.

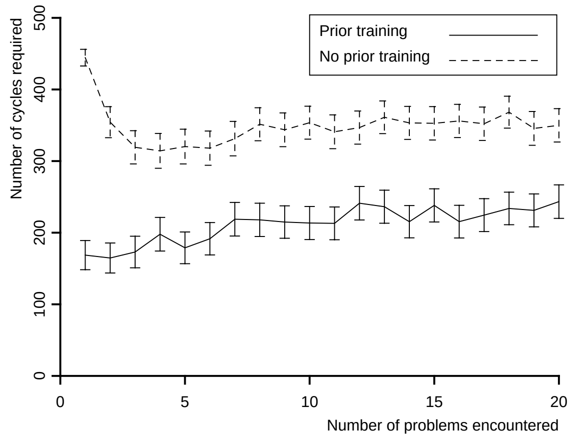

Fig. 5. Execution cycles required to solve a 20 card FreeCell problem as a function of the number of training tasks with and without prior training on simpler problems. Each learning curve shows the mean over 300 different training orders and 95 percent confidence intervals.

6 Related Research

The basic framework we have reported in this paper incorporates ideas from a number of traditions. Our representation and organization of knowledge draws directly from the paradigm of logic programming, whereas its utilization in a recognize-act cycle has more in common with production system architectures. The reliance on heuristic search to resolve goal-driven impasses, coupled with the caching of generalized solutions, comes closest to the performance and learning methods used in problem-solving architectures like Soar (Laird, Rosenbloom, & Newell, 1986) and PRODIGY (Minton, 1990). Finally, we have already noted our debt to Nilsson (1994) for the notion of a teleoreactive system.

However, our approach differs from earlier methods for learning from problem solving in the nature of the acquired knowledge. In contrast to Soar and PRODIGY, which create flat control rules, our framework constructs hierarchical logic programs that incorporate nonterminal symbols. Methods for learning macro-operators (e.g., Iba, 1988; Mooney, 1989) have a similar flavor, in that they explicitly specify the order in which to apply operators, but they do not typically support recursive references, nor do they produce reactive skills that can be used in dynamic domains like driving.

Equally important, our learning method differs substantially from earlier techniques used for improving efficiency of problem solvers. These have used either analytical methods that rely on goal regression to collect conditions on control rules or macro-operators, a relational approach to induction like inductive logic programming, or some combined method (e.g., Estlin & Mooney, 1997). Instead, our method transforms traces of successful means-ends search directly into teleoreactive logic programs, determining their preconditions by a simple method that involves neither analysis or induction, as normally defined.

The cumulative nature of our approach further distinguishes it from earlier efforts. There has been remarkably little work on cumulative learning in problem-solving domains. Ruby and Kibler's (1991) SteppingStone learns to solve more difficult problems based on solutions generalized from simpler ones, which it obtains through a mixture of problem reduction and forward-chaining search. A closer relative is Reddy and Tadepalli's (1997) X-Learn, which acquires goal-decomposition rules from a sequence of training exercises. Their system does not include an execution engine, but it generates recursive hierarchical plans in a cumulative manner using a mixture of analytical and relational learning.

Benson's (1995) TRAIL acquires teleoreactive control programs for use in physical environments. However, it utilizes inductive logic programming to determine the conditions on its rules, which focus on individual actions rather than hierarchical structures. Fern et al. (2004) report another approach to learning reactive controllers that trains itself on increasingly complex problems, but that also acquires flat rules for action selection. Stone and Veloso (2000) describe a system that learns a hierarchical controller for playing robotic soccer, but it acquires quite different types of structure at each level of description. Other work on cumulative learning deals with tasks other than problem solving and reactive control. Sammut and Banerji's (1986) Marvin learns logical concept definitions that are stated in terms of other concepts, whereas Stracuzzi and Utgoff's (2002) STL algorithm incorporates a similar idea but handles many concepts in parallel.

We should also mention another research paradigm that deals with speeding up the execution of logic programs. For instance, Zelle and Mooney (1993) report one such system that combines ideas from explanation-based learning and inductive logic programming to infer the conditions under which clauses should be considered. Work in this area starts and ends with logic programs, whereas our system transforms a weak problem-solving method into an efficient program for reactive control. In summary, although our learning technique incorporates ideas from earlier frameworks, it remains distinct on a number of dimensions.

7 Concluding Remarks

In the preceding pages, we proposed a new representation of knowledge – teleoreactive logic programs – and described how they can be executed over time to control physical agents. In addition, we explained how a means-ends problem solver can utilize them to solve novel tasks and, more important, transform the traces of problem solutions into new clauses that can be executed efficiently. The responsible learning method, goal-driven composition, bears little resemblance to previous techniques, and it acquires recursive, executable skills. We reported experiments that demonstrated the method's ability to learn both reactive driving skills and logic programs for two recursive domains, along with its capacity to benefit from training on tasks of increasing difficulty.

Despite the promise of this new approach to representing, utilizing, and learning knowledge for physical agents, our work remains in its early stages. Future research should demonstrate the acquisition of more complex skills in the driving domain. This will require adding the ability to chain backward off the start conditions of learned clauses, which the problem solver currently avoids. In addition, our method fares well on the domains reported here, but we have observed slight overgeneralization on the Tower of Hanoi, where it acquires a recursive strategy that does not distinguish between the goal and other peg. The learned program has a 50 percent chance of making the wrong choice, which becomes apparent when it runs to completion without reaching the goal. We can avoid this problem by adding some lookahead ability, as in work on hierarchical task networks (e.g., Erol et al., 1994), which have very similar structure. This will require additional effort, but still far less than solving the problem with means-ends analysis.

We should note that, although our approach learns recursive logic programs that generalize to different numbers of objects, its treatment of goals is less flexible. For example, it can acquire a general program for clearing a block that does not depend on the number of others involved, but it cannot learn a program for constructing a tower with arbitrarily specified components. Extending the method's ability to learn about recursive goal structures is an important direction for future research. We should also finds ways to decrease the method's reliance on initial concepts, which it uses to index and organize learned clauses. One approach involves defining a new concept for the start condition of each created clause, which would then be available to support future learning.

In conclusion, our work on learning teleoreactive logic programs is still in its early stages, but it appears to provide a novel and quite promising path to the acquisition of effective control systems that differs significantly from earlier research in this area. We hope to present reports on our progress in future conferences on approaches to learning relational knowledge.

Acknowledgements

This research was funded in part by Grant HR0011-04-1-0008 from DARPA IPTO and by Grant IIS-0335353 from the National Science Foundation. Discussions with Kirstin Cummings, Glenn Iba, Seth Rogers, and Stephanie Sage contributed to the ideas we have presented in this paper.

References

Benson, S. (1995). Induction learning of reactive action models. *Proceedings of the Twelfth International Conference on Machine Learning* (pp. 47–54). San Francisco: Morgan Kaufmann.

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.

Erol, K., Hendler, J., & Nau, D. S. (1994). HTN planning: Complexity and expressivity. *Proceedings of the Twelfth National Conference on Artificial Intelligence* (pp. 1123–1128). Seattle: MIT Press.

Estlin, T. A., & Mooney, R. J. (1997). Learning to improve both efficiency and quality of planning. *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence* (pp. 1227–1232). Nagoya, Japan.

Fern, A., Yoon, S. W., & Givan, R. (2004). Learning domain-specific control knowledge from random walks. *Proceedings of the Fourteenth International Conference on Automated Planning and Scheduling* (pp. 191–199). Whistler, BC: AAAI Press.

Iba, G. A. (1989). A heuristic approach to the discovery of macro-operators. *Machine Learning*, *3*, 285–317.

Laird, J. E., Rosenbloom, P. S., & Newell, A. (1986). Chunking in Soar: The anatomy of a general learning mechanism. *Machine Learning*, *1*, 11–46.

Minton, S. N. (1990). Quantitative results concerning the utility of explanation-based learning. *Artificial Intelligence*, *42*, 363–391.

Mooney, R. J. (1989). The effect of rule use on the utility of explanation-based learning. *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence* (pp. 725–730). Detroit: Morgan Kaufmann.

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.

Nilsson, N. (1994). Teleoreactive programs for agent control. *Journal of Artificial Intelligence Research*, *1*, 139–158.

Reddy, C., & Tadepalli, P. (1997). Learning goal-decomposition rules using exercises. *Proceedings of the Fourteenth International Conference on Machine Learning* (pp. 278–286). San Francisco: Morgan Kaufmann.

Ruby, D., & Kibler, D. (1991). SteppingStone: An empirical and analytical evaluation. *Proceedings of the Tenth National Conference on Artificial Intelligence* (pp. 527–532). Menlo Park, CA: AAAI Press.

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

Sammut, C., & Banerji, R. B. (1986). Learning concepts by asking questions. In R. S. Michalski, J. G. Carbonell, & T. M. Mitchell (Eds.), *Machine learning: An artificial intelligence approach* (Vol. 2). Los Altos, CA: Morgan Kaufmann.

Shavlik, J. W. (1989). Acquiring recursive concepts with explanation-based learning. *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence* (pp. 688–693). Detroit, MI: Morgan Kaufmann.

Stone, P., & Veloso, M. M. (2000). Layered learning. *Proceedings of the Eleventh European Conference on Machine Learning* (pp. 369–381). Barcelona. Springer-Verlag.

Sutton, R. S. & Barto, A. G. (1998). *Reinforcement learning*. Cambridge, MA: MIT Press.

Utgoff, P., & Stracuzzi, D. (2002). Many-layered learning. *Proceedings of the Second International Conference on Development and Learning* (pp. 141–146).

Zelle, J. M., & Mooney, R. J. (1993). Combining FOIL and EBG to speed up logic programs. *Proceedings of the Thirteenth International Joint Conference on Artificial Intelligence* (pp. 1106–1111). Chambery, France: Morgan Kaufmann.