Journal of Machine Learning Research 7 (2006) 493–518 Submitted 7/05; Published 3/06

Learning Recursive Control Programs from Problem Solving

Pat Langley LANGLEY@CSLI.STANFORD.EDU
Dongkyu Choi DONGKYUC@STANFORD.EDU
Computational Learning Laboratory
Center for the Study of Language and Information
Stanford University
Stanford, CA 94305–4115 USA

Editors: Roland Olsson and Ute Schmid

Abstract

In this paper, we propose a new representation for physical control – teleoreactive logic programs – along with an interpreter that uses them to achieve goals. In addition, we present a new learning method that acquires recursive forms of these structures from traces of successful problem solving. We report experiments in three different domains that demonstrate the generality of this approach. In closing, we review related work on learning complex skills and discuss directions for future research on this topic.

Keywords: teleoreactive control, logic programs, problem solving, skill learning

1. Introduction

Human skills have a hierarchical character, with complex procedures defined in terms of more basic ones. In some domains, these skills are recursive in nature, in that structures are specified in terms of calls to themselves. Such recursive procedures pose a clear challenge for machine learning that deserves more attention than it has received in the literature. In this paper we present one response to this problem that relies on a new representation for skills and a new method for acquiring them from experience.

We focus here on the task of learning controllers for physical agents. 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 represent skills as *teleoreactive logic programs*, a formalism that incorporates ideas from logic programming, reactive control, and hierarchical task networks. This framework can encode hierarchical and recursive procedures that are considerably more complex than those usually studied in research on reinforcement learning (Sutton & Barton, 1998) and behavioral cloning (Sammut, 1996), but they can still be executed in a reactive yet goal-directed manner. As we will see, it also embodies constraints that make the learning process tractable.

We assume that an agent uses hierarchical skills to achieve its goals whenever possible, but also that, upon encountering unfamiliar tasks, it falls back on problem solving. The learner begins with primitive skills for the domain, including knowledge of their applicability conditions and their effects, which lets it compose them to form candidate solutions. When the system overcomes such an impasse successfully, which may require substantial search, it learns a new skill that it stores in memory for use on future tasks. Thus, skill acquisition is incremental and intertwined with problem solving. Moreover, learning is cumulative in that skills acquired early on form the building blocks for those mastered later. We have incorporated our assumptions about representation, performance, and learning into ICARUS, a cognitive architecture for controlling physical agents.

Any approach to acquiring hierarchical and recursive procedures from problem solving must address three issues. These concern identifying the hierarchical organization of the learned skills, determining when different skills should have the same name or head, and inferring the conditions under which each skill should be invoked. To this end, our approach to constructing teleoreactive logic programs incorporates ideas from previous work on learning and problem solving, but it also introduces some important innovations.

In the next section, we specify our formalism for encoding initial and learned knowledge, along with the performance mechanisms that interpret them to 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 evidence that the method can learn control programs in three recursive domains, as well as use them on tasks that are more complex than those on which they were acquired. We conclude by reviewing related work on learning and proposing some important directions for additional research.

2. Teleoreactive Logic Programs

As we have noted, our approach revolves around a representational formalism for the execution of complex procedures – teleoreactive logic programs. 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 interleaved knowledge bases. One specifies a set of concepts that the agent uses to recognize classes of situations in the environment and describe them at higher levels of abstraction. These monotonic inference rules have the same semantics as clauses in Prolog 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. In this paper, we assume that a given head appears in only one clause, thus constraining definitions to be conjunctive, although the formalism itself allows disjunctive concepts.

ICARUS distinguishes between primitive conceptual clauses, which refer only to percepts that the agent can observe in the environment, and complex clauses, which refer to other concepts in their bodies. Specific percepts play the same role as ground literals in traditional logic programs, but, because they come from the environment and change over time, we do not consider them part of the program. Table 1 presents some concepts from the Blocks World. Concepts like *unstackable* and *pickupable* are defined in terms of the concepts *clear*, *on*, *ontable*, and *hand-empty*, the subconcept *clear* is defined in terms of *on*, and *on* is defined using two cases of the percept *block*, along with arithmetic tests on their attributes.

A second knowledge base contains a set of skills that the agent can execute in the world. Each skill clause includes a head (a predicate with zero or more arguments) and a body that specifies a set of start conditions and one or more components. Primitive clauses have a single start condition (often a nonprimitive concept) and refer to executable actions that alter the environment. They also

```
((on ?block1 ?block2)
 :percepts  ((block ?block1 xpos ?x1 ypos ?y1)
             (block ?block2 xpos ?x2 ypos ?y2 height ?h2))
 :tests     ((equal ?x1 ?x2) (>= ?y1 ?y2) (<= ?y1 (+ ?y2 ?h2))))
((ontable ?block ?table)
 :percepts  ((block ?block xpos ?x1 ypos ?y1)
             (table ?table xpos ?x2 ypos ?y2 height ?h2))
 :tests     ((>= ?y1 ?y2) (<= ?y1 (+ ?y2 ?h2))))
((clear ?block)
 :percepts  ((block ?block))
 :negatives ((on ?other ?block)))
((holding ?block)
 :percepts  ((hand ?hand status ?block)
             (block ?block)))
((hand-empty)
 :percepts  ((hand ?hand status ?status))
 :tests     ((eq ?status empty)))
((three-tower ?b1 ?b2 ?b3 ?table)
 :percepts  ((block ?b1) (block ?b2) (block ?b3) (table ?table))
 :positives ((on ?b1 ?b2) (on ?b2 ?b3) (ontable ?b3 ?table)))
((unstackable ?block ?from)
 :percepts  ((block ?block) (block ?from))
 :positives ((on ?block ?from) (clear ?block) (hand-empty)))
((pickupable ?block ?from)
 :percepts  ((block ?block) (table ?from))
 :positives ((ontable ?block ?from) (clear ?block) (hand-empty)))
((stackable ?block ?to)
 :percepts  ((block ?block) (block ?to))
 :positives ((clear ?to) (holding ?block)))
((putdownable ?block ?to)
 :percepts  ((block ?block) (table ?to))
 :positives ((holding ?block)))
```

Table 1: Examples of concepts from the Blocks World.

specify the effects of their execution, stated as literals that hold after their completion, and may state requirements that must hold during their execution. Table 2 shows the four primitive skills for the Blocks World, which are similar in structure and spirit to STRIPS operators, but may be executed in a durative manner.

In contrast, nonprimitive 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 depend on the number of levels of recursion, nor do they state requirements. However, the head of each complex skill refers to some concept that the skill aims to achieve, an assumption Reddy and Tadepalli

```
((unstack ?block ?from)
 :percepts  ((block ?block ypos ?y)
             (block ?from))
 :start     ((unstackable ?block ?from))
 :actions   ((*grasp ?block) (*move-up ?block ?y))
 :effects   ((clear ?from)
             (holding ?block)))
((pickup ?block ?from)
 :percepts  ((block ?block ypos ?y)
             (table ?from))
 :start     ((pickupable ?block ?from))
 :actions   ((*grasp ?block) (*move-up ?block ?y))
 :effects   ((holding ?block)))
((stack ?block ?to)
 :percepts  ((block ?block)
             (block ?to xpos ?x ypos ?y height ?height))
 :start     ((stackable ?block ?to))
 :actions   ((*move-over ?block ?x)
             (*move-down ?block (+ ?y ?height))
             (*ungrasp ?block))
 :effects   ((on ?block ?to)
             (hand-empty)))
((putdown ?block ?to)
 :percepts  ((block ?block)
             (table ?to ypos ?y height ?height))
 :start     ((putdownable ?block ?to))
 :actions   ((*move-sideways ?block)
             (*move-down ?block (+ ?y ?height))
             (*ungrasp ?block))
 :effects   ((ontable ?block ?to)
             (hand-empty)))
```

Table 2: Primitive skills for the Blocks World domain. Each 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.

(1997) have also made in their research on task decomposition. This connection between skills and concepts constitutes a key difference between the current approach and our earlier work on hierarchical skills in ICARUS (Choi et al., 2004; Langley & Rogers, 2004), and it figures centrally in the learning methods we describe later. Table 3 presents some recursive skills for the Blocks World, including two clauses for achieving the concept *clear*.

Teleoreactive logic programs are closely related to Nau et al.'s SHOP (1999) formalism for hierarchical task networks. This organizes knowledge into tasks, which serve as heads of clauses, and methods, which specify how to decompose tasks into subtasks. Primitive methods describe the effects of basic actions, much like STRIPS operators. Each method also states its application

```
((clear ?B) 1                          ((unstackable ?B ?A) 3
 :percepts  ((block ?C) (block ?B))     :percepts  ((block ?A) (block ?B))
 :start     ((unstackable ?C ?B))       :start     ((on ?B ?A) (hand-empty))
 :skills    ((unstack ?C ?B)))          :skills    ((clear ?B) (hand-empty)))

((hand-empty) 2                        ((clear ?A) 4
 :percepts  ((block ?C) (table ?T))     :percepts  ((block ?B) (block ?A))
 :start     ((putdownable ?C ?T))       :start     ((on ?B ?A) (hand-empty))
 :skills    ((putdown ?C ?T)))          :skills    ((unstackable ?B ?A)
                                                     (unstack ?B ?A)))
```

Table 3: Some nonprimitive skills for the Blocks World domain that involve recursive calls. Each skill clause has a head that specifies the goal it achieves, a set of typed variables, one or more start conditions, and a set of ordered subskills. Numbers after the head distinguish different clauses that achieve the same goal.

conditions, which may involve predicates that are defined in logical axioms. In our framework, skill heads correspond to tasks, skill clauses are equivalent to methods, and concept definitions play the role of axioms. In this mapping, teleoreactive logic programs are a special class of hierarchical task networks in which nonprimitive tasks always map onto declarative goals and in which top-level goals and the preconditions of primitive methods are always single literals. We will see that these two assumptions play key roles in our approach to problem solving and learning.

Note that every skill/task *S* can be expanded into one or more sequences 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 constraint is weaker than the standard assumption made for macro-operators (e.g., Iba, 1988); it does not guarantee that, once initiated, the sequence will achieve *C*, since other events may intervene or the agent may encounter states in which one of the primitive skills does 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, and we give arguments to this effect at the close of Section 4.

3. Interpreting Teleoreactive Logic Programs

As their name suggests, teleoreactive logic programs are designed for reactive execution in a goal-driven manner, within a physical setting that changes over time. As with most reactive controllers, the associated performance element operates in discrete cycles, but it also involves more sophisticated processing than most such frameworks.

On each decision cycle, ICARUS updates a perceptual buffer with descriptions of all objects that are visible in the environment. Each such percept specifies the object's type, a unique identifier, and zero or more attributes. For example, in the Blocks World these would include structures like *(block A xpos 5 ypos 1 width 1 height 1).* In this paper, we emphasize domains in which the agent perceives the same objects on successive time steps but in which some attributes change value. However, we will also consider teleoreactive systems for domains like in-city driving (Choi et al., 2004) in which the agent perceives different objects as it moves through the environment.

Once the interpreter has updated the perceptual buffer, it invokes an inference module that elaborates on the agent's perceptions. This uses concept definitions to draw logical conclusions from the percepts, which it adds to a conceptual short-term memory. This dynamic store contains higher-level beliefs, cast as relational literals, that are instances of generic concepts. The inference module operates in a bottom-up, data-driven manner that starts from descriptions of perceived objects, such *(block A xpos 5 ypos 1 width 1 height 1)* and *(block B xpos 5 ypos 0 width 1 height 1)*, matches these against the conditions in concept definitions, and infers beliefs about primitive concepts like *(on A B)*. These trigger inferences about higher-level concepts, such as *(clear A)*, which in turn support additional beliefs like *(unstackable A B)*. This process continues until the agent has added all beliefs that are implied by its perceptions and concept definitions.[1]

After the inference module has augmented the agent's perceptions with high-level beliefs, the architecture's execution module inspects this information to decide what actions to take in the environment. To this end, it also examines its current goal, which must be encoded as an instance of some known concept, and its skills, which tell it how to accomplish such goals. Unlike inference, the execution process proceeds in a top-down manner, finding paths through the skill hierarchy that terminate in primitive skills with executable actions. We define a *skill path* to be a chain of skill instances that starts from the agent's goal and descends through the hierarchy along subskill links, unifying the arguments of each subskill consistently with those of its parent.

Furthermore, the execution module only considers skill paths that are *applicable*. This holds if no concept instance that corresponds to a goal along the path is 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 condition is 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. To this end, the module retains the path through the skill hierarchy selected on the previous time step, along with the variable bindings needed to reconstruct it.

For example, imagine a situation in which the block C is on B, B is on A, and A is on the table, in which the goal is *(clear A)*, and in which the agent knows the primitive skills in Table 2 and the recursive skills in Table 3. Further assume that this is the first cycle, so that no previous activities are under way. In this case, the only path through the skill hierarchy is *[(clear A) 4], [(unstackable B A) 3], [(clear B) 1], [(unstack C B)]*. Applying the primitive skill *(unstack C B)* produces a new situation that leads to new inferences, and in which the only applicable path is *[(clear A) 4], [(unstackable B A) 3], [(hand-empty) 2], [(putdown C T)]*. This enables a third path on the next cycle, *[(clear A) 4], [(unstack B A)]*, which generates a state in which the agent's goal is satisfied. Note that this process operates much like the proof procedure in Prolog, except that it involves activities that extend over time.

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, re-executes them to correct the situation. Second, given a choice between two or more applicable skill paths, it selects the one that shares the most elements from the start of the path executed on the

1. Although this mechanism reasons over structures similar to Horn clauses, its operation is closer in spirit to the elaboration process in Soar (Laird et al., 1986) than to the query-driven reasoning in Prolog.

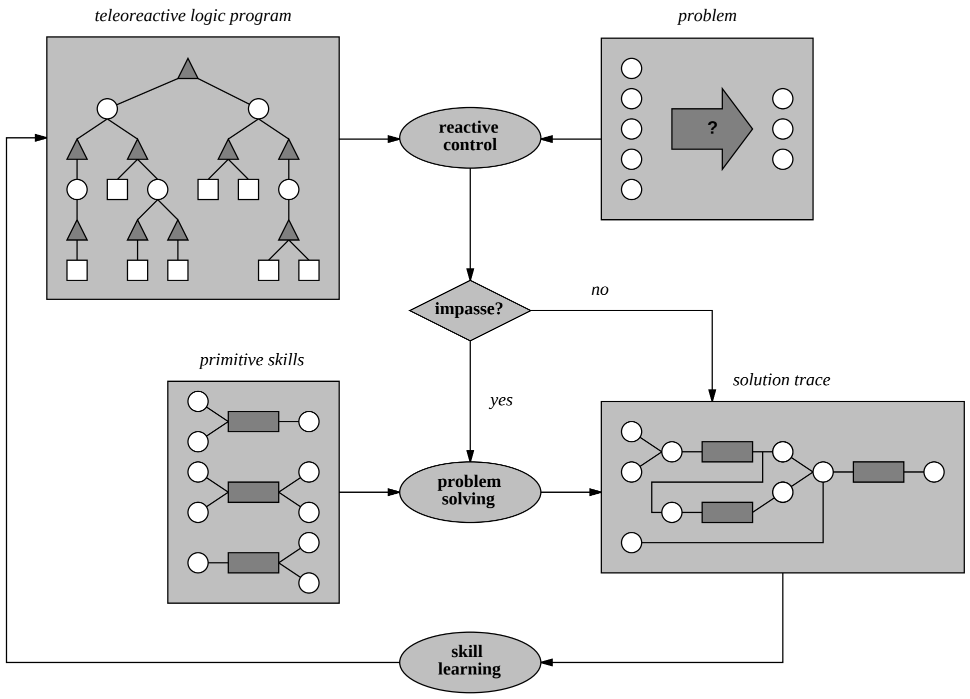

Figure 1: Organization of modules for reactive execution, problem solving, and skill learning, along with their inputs and outputs.

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.

Most research on reactive execution emphasizes dynamic domains in which unexpected events can occur that fall outside the agent's control. Domains like the Blocks World do not have this character, but this does not mean one cannot utilize a reactive controller to direct behavior (e.g., see Fern et al., 2004). Moreover, we have also demonstrated (Choi et al., 2004) the execution module's operation in the domain of in-city driving, which requires reactive response to an environment that changes dynamically. Our framework is relevant to both types of settings.

To summarize, ICARUS' procedure for interpreting teleoreactive logic programs relies on two interacting processes – conceptual inference and skill execution. On each cycle, the architecture perceives objects and infers instances of conceptual relations that they satisfy. After this, it starts from the current goal and uses these beliefs to check the conditions on skill instances to determine which paths are applicable, which in turn constrains the 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 concept that corresponds to the agent's top-level goal is satisfied, when it halts.

4. Solving Problems and Learning Skills

Although one can construct teleoreactive logic programs manually, this process is time consuming and prone to error. Here we report an approach to learning such programs whenever the agent encounters a problem or subproblem that its current skills do not cover. In such cases, the architecture attempts to solve the problem by composing its primitive skills in a way that achieves the goal. Typically, this problem-solving process requires search and, given limited computational resources, may fail. However, when the effort is successful the agent produces a trace of the solution in terms of component skills that achieved the problem's goal. The system transforms this trace into new skill clauses, which it adds to memory for use on future tasks.

Figure 1 depicts this overall organization. As in some earlier problem-solving architectures like PRODIGY (Minton, 1988) and Soar (Laird et al., 1986), problem solving and learning are tightly linked and both are driven by impasses. A key difference is that, in these systems, learning produces search-control knowledge that makes future problem solving more effective, whereas in our framework it generates teleoreactive logic programs that the agent uses in the environment. Nevertheless, there remain important similarities that we discuss later at more length.

4.1 Means-Ends Problem Solving

As described earlier, the execution module selects skill clauses that should achieve the current goal and that have start conditions which match its current beliefs about the environment. Failure to retrieve such a clause produces an impasse that leads the architecture to invoke its problem-solving module. Table 4 presents pseudocode for the problem solver, which utilizes a variant of means-ends analysis (Newell & Simon, 1961) that chains backward from the goal. This process relies on a goal stack that stores both subgoals and skills that might accomplish them. The top-level goal is simply the lowest element on this stack.

Despite our problem-solving method's similarity to means-ends analysis, it differs from standard formulation in three important ways:

- whenever the skill associated with the topmost goal on the stack becomes applicable, the system executes it in the environment, which leads to tight interleaving of problem solving and control;
- both the start conditions of primitive skills (i.e., operators) and top-level goals must be cast as single relational literals, which may be defined concepts;[2]
- backward chaining can occur not only off the start condition of primitive skills but also off the definition of a concept, which means the single-literal assumption causes no loss of generality.

As we will see shortly, the second and third of these assumptions play key roles in the mechanism for learning new skills, but we should first examine the operation of the problem-solving process itself.

As Table 4 indicates, the problem solver pushes the current goal G onto the goal stack, then checks it on each execution cycle to determine whether it has been achieved. If so, then the module pops the stack and focuses on G's parent goal or, upon achieving the top-level goal, simply halts. If the current goal G is not satisfied, then the architecture retrieves all nonprimitive skills with heads that unify with G and, if any participate in applicable paths through the skill hierarchy, selects the first one found and executes it. This execution may require many cycles, but eventually it produces a new environmental state that either satisfies G or constitutes another impasse.

If the problem solver cannot find any complex skills indexed by the goal G, it instead retrieves all primitive skills that produce G as one of their effects. The system then generates candidate instances of these skills by inserting known objects as their arguments. To select among these skill instances, it expands the instantiated start condition of each skill instance to determine how many of its primitive components are satisfied, then selects the one with the fewest literals unsatisfied in the current situation. If the candidates tie on this criterion, then it selects one at random. If the selected

2. We currently define all concepts manually, but it would not be difficult to have the system define them automatically for operator preconditions and conjunctive goals.

```
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.
     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.
                    Push the start condition C of Q onto goal stack GS.
               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.
                         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.
```

Table 4: Pseudocode for interleaving means-ends problem solving with skill execution.

skill instance's condition is met, the system executes the skill instance in the environment until it achieves the associated goal, which it then pops from the stack. If the condition is not satisfied, the architecture makes it the current goal by pushing it onto the stack.

However, if the problem solver cannot find any skill clause that would achieve the current goal G, it uses G's concept definition to decompose the goal into subgoals. If more than one subgoal is unsatisfied, the system selects one at random and calls the problem solver on it recursively, which makes it the current goal by pushing it onto the stack. This leads to chaining off the start condition of additional skills and/or the definitions of other concepts. Upon achieving a subgoal, the architecture pops the stack and, if other subconcepts remain unsatisfied, turns its attention to achieving them. Once all have been satisfied, this means the parent goal G has been achieved, so it pops the stack again and focuses on the parent.

Of course, the problem-solving module must make decisions about which skills to select during skill chaining and the order in which it should tackle subconcepts during concept chaining. The system may well make the incorrect choice at any point, which can lead to failure on a given subgoal when no alternatives remain or when it reaches the maximum depth of the goal stack. In such cases, it pops the current goal, stores the failed candidate with its parent goals to avoid considering them in the future, and backtracks to consider other options. This strategy produces depth-first search through the problem space, which can require considerable time on some tasks.

Figure 2 shows an example of the problem solver's behavior on the Blocks World in a situation where block A is on the table, block B is on A, block C is on B, and the hand is empty. Upon being given the objective *(clear A)*, the architecture 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. In this case,

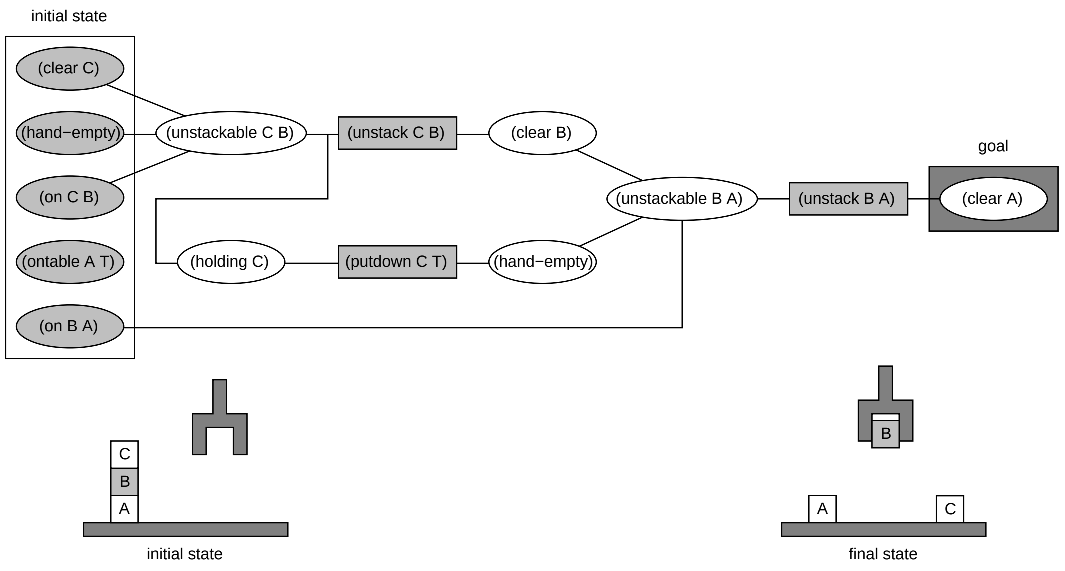

Figure 2: A trace of successful problem solving in the Blocks World, which ellipses indicating concepts/goals and rectangles denoting primitive skills.

invoking the primitive skill instance *(unstack B A)* would produce the desired result. However, this cannot yet be applied because its instantiated start condition, *(unstackable B A)*, does not hold, so the system stores the skill instance with the initial goal and pushes this subgoal onto the stack.

Next, the problem solver attempts to retrieve skills that would achieve *(unstackable B A)* but, because it has no such skills in memory, it resorts to chaining off the definition of *unstackable*. This involves three instantiated subconcepts – *(clear)*, *(on B A)*, and *(hand-empty)* – but only the first of these is unsatisfied, so the module pushes this onto the goal stack. In response, it considers skills that would produce this literal as an effect and retrieves the skill instance *(unstack C B)*, which it stores with the current goal.

In this case, the start condition of the selected skill, *(unstackable C B)*, already holds, so the architecture executes *(unstack C B)*, which alters the environment and causes the agent to infer *(clear B)* from its percepts. In response, it pops this goal from the stack and reconsiders its parent, *(unstackable B A)*. Unfortunately, this has not yet been achieved because executing the skill has caused the third of its component concept instances, *(hand-empty)*, to become false. Thus, the system pushes this onto the stack and, upon inspecting memory, retrieves the skill instance *(putdown C T)*, which it can and does execute.

This second step achieves the subgoal *(hand-empty)*, which in turn lets the agent infer *(unstackable B A)*. Thus, the problem solver pops this element from the goal stack and executes the skill instance it had originally selected, *(unstack B A)*, in the new situation. Upon completion, the system perceives that the altered environment satisfies the top-level goal, *(clear A)*, which leads it to halt, since it has solved the problem. Both our textual description and the graph in Figure 2 represent the trace of successful problem solving; as noted earlier, finding such a solution may well involve search, but we have omitted missteps that require backtracking for the sake of clarity.

Despite the clear evidence that humans often resort to means-ends analysis when they encounter novel problems (Newell & Simon, 1961), this approach to problem solving has been criticized in the AI planning community because it searches over a space of totally ordered plans. As a result, on problems for which the logical structure of a workable plan is only partially ordered, it can carry out extra work by considering alternative orderings that are effectively equivalent. However, the method also has clear advantages, such as low memory load because it must retain only the current stack rather than a partial plan. Moreover, it provides direct support for interleaving of problem solving and execution, which is desirable for agents that must act in their environment.

Of course, executing a component skill before it has constructed a complete plan can lead the system into difficulty, since the agent cannot always backtrack in the physical world and can produce situations from which it cannot recover without starting over on the problem. In such cases, the problem solver stores the goal for which the executed skill caused trouble, along with everything below it in the stack. The system begins the problem again, this time avoiding the skill and selecting another option. If a different execution error occurs this time, the module again stores the problematic skill and its context, then starts over once more. In this way, the architecture continues to search the problem space until it achieves its top-level goal or exceeds the number of maximum allowed attempts.[3]

4.2 Goal-Driven Composition of Skills

Any method for learning teleoreactive logic programs or similar structures must address three issues. First, it must determine the structure of the hierarchy that decomposes problems into subproblems. Second, the technique must identify when different clauses should have the same head and thus be considered in the same situations. Finally, it must infer the conditions under which to invoke each clause. The approach we describe here relies on results produced by the problem solver to answer these questions. Just as problem solving occurs whenever the system encounters an impasse, that is, a goal it cannot achieve by executing stored skills, so learning occurs whenever the system resolves an impasse by successful problem solving. The ICARUS architecture shares this idea with earlier frameworks like Soar and PRODIGY, although the details differ substantially.

The response to the first issue is that *hierarchical structure is determined by the subproblems handled during problem solving*. As Figure 2 illustrates, this takes the form of a semilattice in which each subplan has a single root node. This structure follows directly from our assumptions that each primitive skill has one start condition and each goal is cast as a single literal. Because the problem solver chains backward off skill and concept definitions, the result is a hierarchical structure that suggests a new skill clause for each subgoal. Table 5 (a) presents the clauses that the system proposes based on the solution to the *(clear A)* problem, without specifying their heads or conditions. Figure 2 depicts the resulting hierarchical structure, using numbers to indicate the order in which the system generates each clause.

The answer to the second question is that *the head of a learned skill clause is the goal literal that the problem solver achieved for the subproblem that produced it*. This follows from our assumption that the head of each clause in a teleoreactive logic program specifies some concept that the clause will produce if executed. At first glance, this appears to confound skills with concepts, but another view is that it indexes skill clauses by the concepts they achieve. Table 5 (b) shows the clauses learned from the problem-solving trace in Figure 2 once the heads have been inserted. Note that this

3. The problem solver also starts over if it has not achieved the top-level objective within a given number of cycles. Jones and Langley (in press) report another variant of means-ends problem solving that uses a similar restart strategy but keeps no explicit record of previous failed paths.

```
(a) (<head> 1                          (<head> 3
     :percepts ((block ?C) (block ?B))   :percepts ((block ?A) (block ?B))
     :start    <conditions>              :start    <conditions>
     :skills   ((unstack ?C ?B)))        :skills   ((clear ?B) (hand-empty)))

    (<head> 2                          (<head> 4
     :percepts ((block ?C) (table ?T))   :percepts ((block ?B) (block ?A))
     :start    <conditions>              :start    <conditions>
     :skills   ((putdown ?C ?T)))        :skills   ((unstackable ?B ?A)
                                                     (unstack ?B ?A)))
----------------------------------------------------------------------------
(b) ((clear ?B) 1                      ((unstackable ?B ?A) 3
     :percepts ((block ?C) (block ?B))   :percepts ((block ?A) (block ?B))
     :start    <conditions>              :start    <conditions>
     :skills   ((unstack ?C ?B)))        :skills   ((clear ?B) (hand-empty)))

    ((hand-empty) 2                    ((clear ?A) 4
     :percepts ((block ?C) (table ?T))   :percepts ((block ?B) (block ?A))
     :start    <conditions>              :start    <conditions>
     :skills   ((putdown ?C ?T)))        :skills   ((unstackable ?B ?A)
                                                     (unstack ?B ?A)))
----------------------------------------------------------------------------
(c) ((clear ?B) 1                      ((unstackable ?B ?A) 3
     :percepts ((block ?C) (block ?B))   :percepts ((block ?A) (block ?B))
     :start    ((unstackable ?C ?B))     :start    ((on ?B ?A) (hand-empty))
     :skills   ((unstack ?C ?B)))        :skills   ((clear ?B) (hand-empty)))

    ((hand-empty) 2                    ((clear ?A) 4
     :percepts ((block ?C) (table ?T))   :percepts ((block ?B) (block ?A))
     :start    ((putdownable ?C ?T))     :start    ((on ?B ?A) (hand-empty))
     :skills   ((putdown ?C ?T)))        :skills   ((unstackable ?B ?A)
                                                     (unstack ?B ?A)))
```

Table 5: Skill clauses for the Blocks World learned from the trace in Figure 2 (a) after hierarchical structure has been determined, (b) after the heads have been identified, and (c) after the start conditions have been inserted. Numbers after the heads indicate the order in which clauses are generated.

strategy leads directly to the creation of recursive skills whenever a conceptual predicate *P* is the goal and *P* also appears as a subgoal. In this example, because *(clear A)* is the top-level goal and *(clear B)* occurs as a subgoal, one of the clauses learned for *clear* is defined recursively, although this happens indirectly through *unstackable*.

Clearly, introducing recursive statements can easily lead to overly general or even nonterminating programs. Our approach avoids the latter because the problem solver never considers a subgoal if it already occurs earlier in the goal stack; this ensures that subgoals which involve the same predicate always have different arguments. However, we still require some means to address the third issue of determining conditions on learned clauses that guards against the danger of overgen-

```
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 create a new skill clause N with head G,
              with the head of $S_2$ as the only subskill,
              and with the same start condition as $S_2$.
            Return the literal for skill clause N.
       Else create a new skill clause N with head G,
              with the heads of $S_1$ and $S_2$ as ordered subskills,
              and with the same start condition as $S_1$.
            Return the literal for skill clause N.
  Else if the goal G involves concept chaining,
       Then let $C_1$, ..., $C_k$ be G’s initially satisfied subconcepts.
            Let $C_{k+1}$, ..., $C_n$ 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 $C_1$, ..., $C_k$ as start conditions.
            Return the literal for skill clause N.
```

Table 6: Pseudocode for creation of skill clauses through goal-driven composition.

eralization. The response differs depending on whether the problem solver resolves an impasse by chaining backward on a primitive skill or by chaining on a concept definition.

Suppose the agent achieves a subgoal G through skill chaining, say by first applying skill S_1 to satisfy the start condition for S_2 and executing the skill S_2, producing a clause with head G and ordered subskills S_1 and S_2. In this case, *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 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.[4]

In contrast, suppose the agent achieves a goal concept G through concept chaining by satisfying the subconcepts $G_{k+1},\ldots,G_n$, in that order, while subconcepts $G_1,\ldots,G_k$ were true at the outset. In response, the system would construct a new skill clause with head G and the ordered subskills $G_{k+1},\ldots,G_n$, each of which the system already knew and used to achieve the associated subgoal or which it learned from the successful solution of one of the subproblems. 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 $G_1,\ldots,G_k$ are not satisfied, in which case the sequence $G_{k+1},\ldots,G_n$ may not achieve the goal G. Table 6 gives pseudocode that summarizes both methods for determining the conditions on new clauses.

Table 5 (c) presents the conditions learned for each of the skill clauses learned from the trace in Figure 2. Two of these (clauses 1 and 2) are trivial because they result from degenerate subproblems that the system solves by chaining off a single primitive operator. Another skill clause (3) is more

4. If skill S_2 is executed without invoking another skill to meet its start condition, the method creates a new clause, with S_2 as its only subskill, that restates the original skill in a new form with G in its head.

```
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.
                    Push the start condition C of Q onto goal stack GS.
                    Store Q with goal G as its last subskill.
                    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.
```

Table 7: Pseudocode for interleaved problem solving and execution extended to support goal-driven composition of skills. New steps are indicated in italic font.

interesting because it results from chaining off the concept definition for *unstackable*. This has the start conditions *(on ?A ?B)* and *(hand-empty)* because the subconcept instances *(on A B)* and *(hand-empty)* held at the outset.[5] The final clause (4) is most intriguing because it results from using a learned clause (3) followed by the primitive skill instance *(unstack B A)*. In this case, the start condition is the same as that for the first subskill clause (3).

Upon initial inspection, the start conditions for clause 3 for achieving *unstackable* may appear overly general. However, recall that the skill clauses in a teleoreactive logic program are interpreted not in isolation but as parts of chains through the skill hierarchy. The interpreter will not select a path for execution unless all conditions along the path from the top clause to the primitive skill are satisfied. This lets the learning method store very abstract conditions for new clauses with less danger of overgeneralization. On reflection, this scheme is the only one that makes sense for recursive control programs, since static preconditions cannot characterize such structures. Rather,

5. Although primitive skills have only one start condition, we do not currently place this constraint on learned clauses, as they are not used in problem solving and it makes acquired programs more readable.

the architecture must compute appropriate preconditions dynamically, depending on the depth of recursion. The Prolog-like interpreter used for skill selection provides this flexibility and guards against overly general behavior.

We refer to the learning mechanism that embodies these answers as *goal-driven composition*. This process 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. The technique shares this characteristic with analytical methods for learning from problem solving, such as those found in Soar and PRODIGY. But unlike these methods, it learns hierarchical skills that decompose problems into subproblems, and, unlike most methods for forming macro-operators, it acquires disjunctive and recursive skills. Moreover, learning is cumulative in that skills learned from one problem are available for use on later tasks. Taken together, these features make goal-driven composition a simple yet powerful approach to learning logic programs for reactive control. Nor is the method limited to working with means-ends analysis; it should operate over traces of any planner that chains backward from a goal.

The architecture's means-ends module must retain certain information during problem solving to support the composition of new skill clauses. Table 7 presents expanded pseudocode that specifies this information and when the system stores it. The form and content is similar to that recorded in Veloso and Carbonell's (1993) approach to derivational analogy. The key difference is that their system stores details about subgoals, operators, and preconditions in specific cases that drive future problem solving, whereas our approach transforms these instances into generalized hierarchical structures for teleoreactive control.

We should clarify that the current implementation invokes a learned clause only when it is applicable in the current situation, so the problem solver never chains off its start conditions. Mooney (1989) incorporated a similar constraint into his work on learning macro-operators to avoid the utility problem (Minton, 1990), in which learned knowledge reduces search but leads to slower behavior. However, we have extended his idea to cover cases in which learned skills can solve subproblems, which supports greater transfer across tasks. In our framework, this assumption means that clauses learned from skill chaining have a left-branching structure, with the second subskill being primitive.

In Section 2, we stated that every skill clause in a teleoreactive logic program can be expanded into one or more sequences of primitive skills, and that each sequence, if executed legally, will produce a state that satisfies the clause's head concept. Here we argue that goal-driven composition learns sets of skill clauses for which this condition holds. As in most research on planning, we assume that the preconditions and effects of primitive skills are accurate, and also that no external forces interfere. First consider a clause with the head H that has been created as the result of successful chaining off a primitive skill. This learned clause is guaranteed to achieve the goal concept H because H must be an effect of its final subskill or the chaining would never have occurred.

Now consider a clause with the head H that has been created as the result of successful chaining off a conjunctive definition of the concept H. This clause describes a situation in which some subconcepts of H hold but others must still be achieved to make H true. Some subconcepts may become unsatisfied in the process and need to be reachieved, but the ordering on subgoals found during problem solving worked for the particular objects involved, and replacing constants with variables will not affect the result. Thus, if the clause's start conditions are satisfied, achieving the subconcepts in the specified order will achieve H. Remember that our method does *not* guarantee, like those for learning macro-operators, that a given clause expansion *will* run to completion. Whether this occurs in a given domain is an empirical question, to which we now turn.

5. Experimental Studies of Learning

As previously reported (Choi & Langley, 2005), the means-ends problem solving and learning mechanisms just described construct properly organized teleoreactive logic programs. After learning, the agent can simply retrieve and execute the acquired programs to solve similar problems without falling back to problem solving. Here we report promising results from more systematic and extensive experiments. The first two studies involve inherently recursive but nondynamic domains, whereas the third involves a dynamic driving task.

5.1 Blocks World

The Blocks World involves an infinitely large table with cubical blocks, along with a manipulator that can grasp, lift, carry, and ungrasp one block at a time. In this domain, we wrote an initial program with nine concepts and four primitive skills. Additionally, we provided a concept for each of four different goals.[6] Theoretically, this knowledge is sufficient to solve any problem in the domain, but the extensive search required would make it intractable to solve tasks with many blocks using only basic knowledge. In fact, only 20 blocks are enough to make the system search for half an hour. Therefore, we wanted the system to learn teleoreactive logic programs that it could execute recursively to solve problems with arbitrary complexity. We have already discussed a recursive program acquired from one training problem, which requires clearing the lowest object in a stack of three blocks, but many other tasks are possible.

To establish that the learned programs actually help the architecture to solve more complex problems, we ran an experiment that compared the learning and non-learning versions. We presented the system with six ten-problem sets of increasing complexity, one after another. More specifically, we used sets of randomly generated problems with 5, 10, 15, 20, 25, and 30 blocks. If the goal-driven composition mechanism is effective, then it should produce noticeable benefits in harder tasks when the learning is active.

We carried out 200 runs with different randomized orders within levels of task difficulty. In each case, we let the system run a maximum of 50 decision cycles before starting over on a problem and attempt a task at most five times before it gave up. For this domain, we set the maximum depth of the goal stack used in problem solving to eight. Figure 3 displays the number of execution cycles and the CPU time required for both conditions, which shows a strong benefits from learning.

With number of cycles as the performance measure, we see a systematic decrease as the system gains more experience. Every tenth problem introduces five additional objects, but the learning system requires no extra effort to solve them. The architecture has constructed general programs that let it achieve familiar goals for arbitrary numbers of blocks without resorting to deliberative problem solving. Inspection reveals that it acquires the nonprimitive skill clauses in Table 3, as well as additional ones that make recursive calls. In contrast, the nonlearning system requires more decision cycles on harder problems, although this levels off later in the curve, as the problem solver gives up on very difficult tasks.

6. These concerned achieving situations in which a given block is clear, one block is on another, one block is on another and a third block is on the table, and three blocks are arranged in a tower.

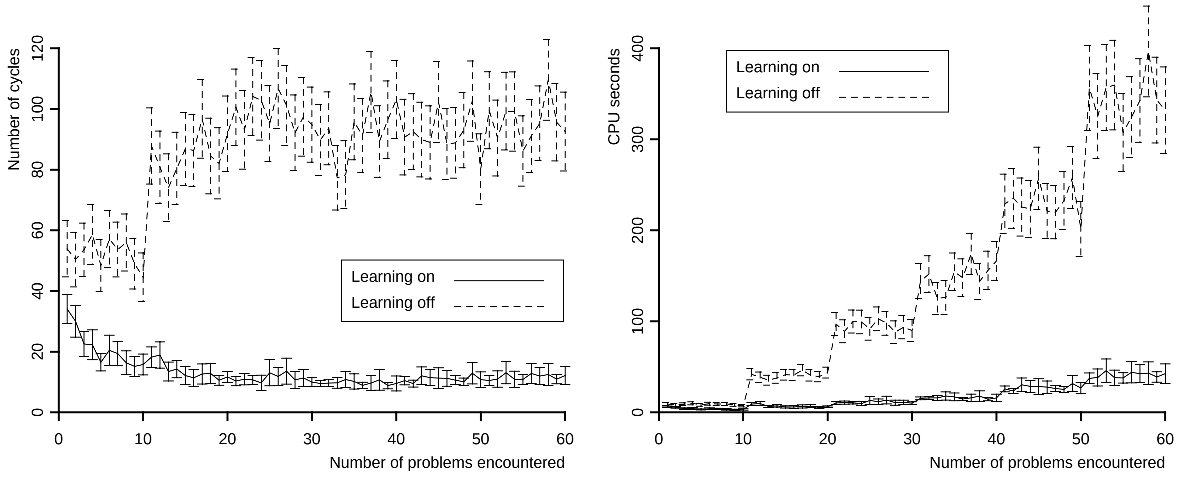

Figure 3: Execution cycles and CPU times required to solve a series of 5, 10, 15, 20, 25, and 30-block problems (10 different tasks at each level) in the Blocks World as a function of the number of tasks with and without learning. Each learning curve shows the mean over 200 different task orders and 95 percent confidence intervals.

The results for solution time show similar benefits, with the learning condition substantially outperforming the condition without. However, the figure also indicates that even the learning version slows down somewhat as it encounters problems with more blocks. Analysis of individual runs suggests this results from the increased cost of matching against objects in the environment, which is required in both the learning and nonlearning conditions. This poses an issue, not for our approach to skill construction but to our architectural framework, so it deserves attention in future research.

Table 8 shows the average results for each level of problem complexity, including the probability that the system can solve a problem within the allowed number of cycles and attempts. In addition to presenting the first two measures at more aggregate levels, it also reveals that, without learning, the chances of finding a solution decrease with the number of blocks in the problem. Letting the system carry out more search would improve these scores, but only at the cost of increasing the number of cycles and CPU time needed to solve the more difficult problems.

5.2 FreeCell Solitaire

FreeCell is a solitaire game with eight columns of stacked cards, all face up and visible to the player, that has been used in AI planning competitions (Bacchus, 2001). There are four free cells, which can hold any single card at a time, and four home cells that correspond to the four different suits. The goal is to move all the cards on the eight columns to the home cells for their suits in ascending order. The player can move only the cards on the top of the eight columns and the ones in the free cells. Each card can be moved to a free cell, to the proper home cell, or to an empty column. In addition, the player can move a card to a column whose top card has the next number and a different color. As in the Blocks World, we provided a simulated environment that allows legal moves and updates the agent's perceptions.

Blocks	Learning			No Learning		
	cycles	CPU	P(sol)	cycles	CPU	P(sol)
5	21.25	4.03	0.997	52.52	8.82	0.958
10	13.61	6.90	0.997	85.15	40.60	0.857
15	11.22	11.13	0.995	98.82	94.93	0.816
20	9.76	16.09	0.997	92.06	149.05	0.863
25	11.04	27.41	0.996	91.77	230.43	0.842
30	11.67	40.85	0.995	95.89	344.49	0.826

Table 8: Aggregate scaling results for the Blocks World.

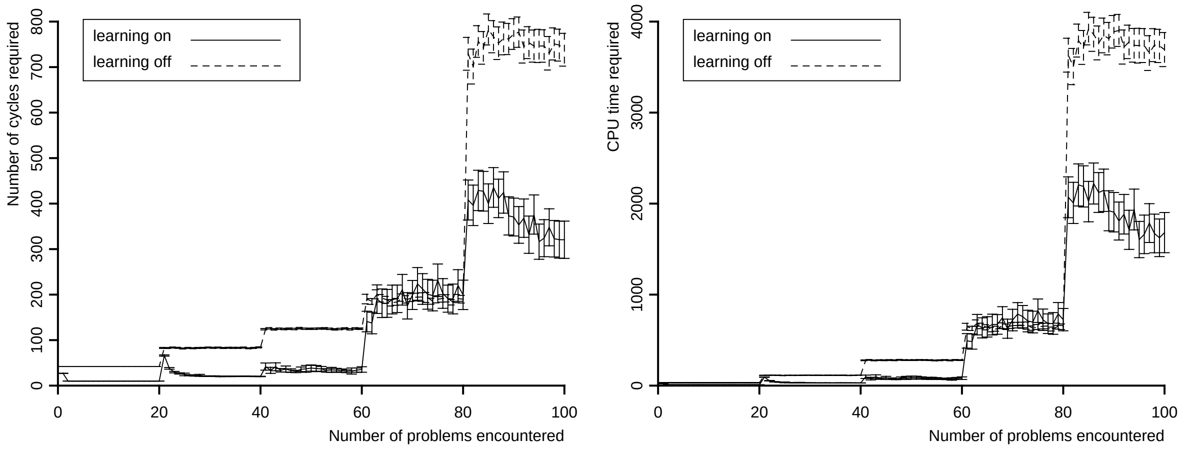

Figure 4: Execution cycles and CPU times required to solve a series of 8, 12, 16, 20, and 24-card FreeCell problems (20 different tasks each) as a function of the number of tasks with and without learning. Each learning curve shows the mean over 300 different task orders and 95 percent confidence intervals.

For this domain, we provided the architecture with an initial program which involves 24 concepts and 12 primitive skills that should, in principle, let it solve any initial configuration with a feasible solution path. (Most but not all FreeCell problems are solvable.) However, the agent may find a solution only after a significant amount of search using its means-ends problem solver. Again we desired the system to learn teleoreactive logic programs that it can execute on complex FreeCell problems with little or no search. In this case, we presented tasks as a sequence of five 20-problem sets with 8, 12, 16, 20, and 24 cards. On each problem, we let the system run at most 1000 decision cycles before starting over, attempt the task no more than five times before halting, and create goal stacks up to 30 in depth. We ran both the learning and nonlearning versions on 300 sets of randomly generated problems and averaged the results. Figure 4 shows the number of cycles and the CPU time required to solve tasks as a function of the number of problems encountered.

In the learning condition, the system rapidly acquired recursive FreeCell programs that reduced considerably the influence of task difficulty as compared to the nonlearning version. As before,

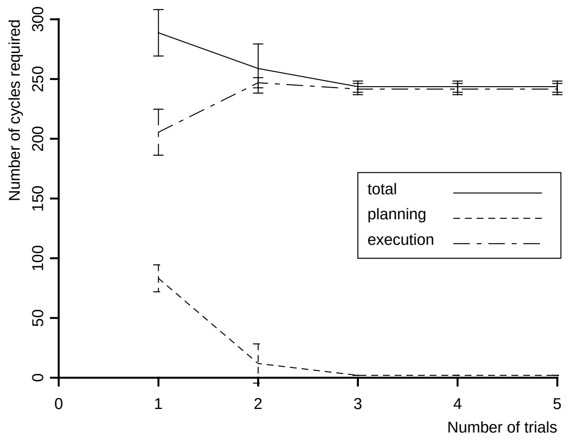

Figure 5: The total number of cycles required to solve a particular right-turn task along with the planning and execution times, as a function of the number of trials. Each learning curve shows the mean computed over ten sets of trials and 95 percent confidence intervals.

the benefits are reflected in both the number of cycles needed to solve problems and in the CPU time. However, increasing the number of cards in this domain can alter the structure of solutions, so the learning system continued to invoke means-ends problem solving in later portions of the curve. For instance, situations with 20 cards often require column-to-column moves that do not appear in simpler tasks, which caused comparable behavior in the two conditions at this complexity level. However, the learning system took advantage of this experience to handle 24-card problems with much less effort. Learning also increased the probability of solution (about 80 percent) over the nonlearning version (around 50 percent) on these tasks.

5.3 In-City Driving

The in-city driving domain involves a medium-fidelity simulation of a downtown driving environment. The city has several square blocks with buildings and sidewalks, street segments, and intersections. Each street segment includes a yellow center line and white dotted lane lines, and it has its own speed limit the agent should observe. Buildings on each block have unique addresses, to help the agent navigate through the city easily and to allow specific tasks like package deliveries. A typical city configuration we used has nine blocks, bounded by four vertical streets and four horizontal streets with four lanes each.

For this domain, we provided the system 41 concepts and 19 primitive skills. With the basic knowledge, the agent can describe its current situation at multiple levels of abstraction and perform actions for accelerating, decelerating, and steering left or right at realistic angles. Thus, it can operate a vehicle, but driving safely in a city environment is a totally different story. The agent must still learn how to stay aligned and centered within lane lines, change lanes, increase or decrease speed for turns, and stop for parking. To encourage such learning, we provided the agent with the task of moving to a destination on a different street segment that requires a right turn. To achieve this task, it resorted to problem solving, which found a solution path that involved changing to the rightmost lane, staying aligned and centered until the intersection, steering right to place the car in the target segment, and finally aligning and centering in the new lane.

We recorded the total number of cycles to solve this task, along with its breakdown into the cycles devoted to planning and to execution, as a function of the number of trials. Figure 5 shows the learning curve that results from averaging over ten different sets of trials. As the system accumulates knowledge about the driving task, its planning effort effectively disappears, which leads to an overall reduction in the total cycles, even though the execution cycles increase slightly. The latter occurs because the vehicle happens to be moving in the right direction at the outset, which accidently brings it closer to the goal while the system is engaged in problem solving. After learning, the agent takes the same actions intentionally, which produces the increase in execution cycles. We should note that this task is dominated by driving time, which places a lower bound on the benefits of learning even when behavior becomes fully automatized.

We also inspected the skills that the architecture learned for this domain. Table 9 shows the five clauses it acquires by the end of a typical training run. These structures include two recursive references, one in which *in-intersection-for-right-turn* invokes itself directly, but also a more interesting one in which *driving-in-segment* calls itself indirectly through *in-segment*, *in-intersection-for-right-turn*, and *in-rightmost-lane*. Testing this teleoreactive logic program on streets with more lanes than occur in the training task suggests that it generalizes correctly to these situations.

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 (Clocksin & Mellish, 1981), whereas its utilization in a recognize-act cycle has more in common with production-system architectures (Neches, Langley, & Klahr, 1987). 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 improving the efficiency of problem solvers 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. Shavlik (1989) reports a system that learns recursive macro-operators but that, like other work in this area, does not acquire reactive controllers.

Moreover, both traditions have used sophisticated analytical methods that rely on goal regression to collect conditions on control rules or macro-operators, nonincremental empirical techniques like inductive logic programming, or combinations of such methods (e.g., Estlin & Mooney, 1997). Instead, goal-driven composition 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, and that operates in an incremental and cumulative fashion.

Previous research on learning for reactive execution, like work on search control, has emphasized unstructured knowledge. For example, Benson's (1995) TRAIL acquires teleoreactive control

```
((driving-in-segment ?me ?g994 ?g1021)
 :percepts ((segment ?g994) (lane-line ?g1021) (self ?me))
 :start    ((in-segment ?me ?g994) (steering-wheel-straight ?me))
 :skills   ((in-lane ?me ?g1021)
            (centered-in-lane ?me ?g994 ?g1021)
            (aligned-with-lane-in-segment ?me ?g994 ?g1021)
            (steering-wheel-straight ?me)))
((driving-in-segment ?me ?g998 ?g1008)
 :percepts ((segment ?g998) (lane-line ?g1008) (self ?me))
 :start ((steering-wheel-straight ?me))
 :skills   ((in-segment ?me ?g998)
            (centered-in-lane ?me ?g998 ?g1008)
            (aligned-with-lane-in-segment ?me ?g998 ?g1008)
            (steering-wheel-straight ?me)))

((in-segment ?me ?g998)
 :percepts ((self ?me) (intersection ?g978) (segment ?g998))
 :start    ((last-lane ?g1021))
 :skills   ((in-intersection-for-right-turn ?me ?g978)
            (steer-for-right-turn ?me ?g978 ?g998)))
((in-intersection-for-right-turn ?me ?g978)
 :percepts ((lane-line ?g1021) (self ?me) (intersection ?g978))
 :start    ((last-lane ?g1021))
 :skills   ((in-rightmost-lane ?me ?g1021)
            (in-intersection-for-right-turn ?me ?g978)))
((in-rightmost-lane ?me ?g1021)
 :percepts ((self ?me) (lane-line ?g1021))
 :start    ((last-lane ?g1021))
 :skills   ((driving-in-segment ?me ?g994 ?g1021)))
```

Table 9: Recursive skill clauses learned for the in-city driving domain.

programs for use in physical environments, but it utilizes inductive logic programming to determine local rules for individual actions rather than hierarchical structures. Fern et al. (2004) report an approach to learning reactive controllers that trains itself on increasingly complex problems, but that also acquires decision lists for action selection. Khardon (1999) describes another method for learning ordered, but otherwise unstructured, control rules from observed problem solutions.

Our approach shares some features with research on inductive programming, which focuses on synthesizing iterative or recursive programs from input-output examples. For instance, Schmid's (2005) IPAL generates an initial program from the results of problem solving by replacing constants with constructive expressions with variables, then transforms it into a recursive program through inductive inference steps. Olsson's (1995) ADATE also generates recursive programs through program refinement transformations, but carries out an iterative deepening search guided by criteria like fit to training examples and syntactic complexity. Schmid's work comes closer to our own, in that both operate over problem-solving traces and generate recursive programs, but our method produces these structures directly, rather than using explicit transformation or revision steps.

Perhaps the closest relative to our approach is Reddy and Tadepalli's (1997) X-Learn, which acquires goal-decomposition rules from a sequence of training problems. Their system does not include an execution engine, but it generates recursive hierarchical plans in a cumulative manner that also identifies declarative goals with the heads of learned clauses. However, because it invokes forward-chaining rather than backward-chaining search to solve new problems, it relies on the trainer to determine program structure. X-Learn also uses a sophisticated mixture of analytical and relational techniques to determine conditions, rather than our much simpler method. Ruby and Kibler's (1991) SteppingStone has a similar flavor, in that it learns generalized decompositions through a mixture of problem reduction and forward-chaining search. Marsella and Schmidt's (1993) system also acquires task-decomposition rules by combining forward and backward search to hypothesize state pairs, which in turn produce rules that it revises after further experience.

Finally, we should mention another research paradigm that deals with speeding up the execution of logic programs. One example comes from Zelle and Mooney (1993), who report a 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 standard 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. Directions for Future Research

Despite the promise of this new approach to representing, utilizing, and learning knowledge for teleoreactive control, our work remains in its early stages. Future research should demonstrate the acquisition of complex skills on additional domains. These should include both classical domains like logistics planning and dynamic settings like in-city driving. We have reported preliminary results on the latter, but our work in this domain to date has dealt with relatively simple skills, such as changing lanes and slowing down to park. Humans' driving knowledge is far more complex, and we should demonstrate that our methods are sufficient to acquire many more of them.

Note that, although driving involves reactive control, it also benefits from route planning and other high-level activities. Recall that our definition of teleoreactive logic programs, and our method for learning them, guarantees only that a skill will achieve its associated goal if it executes successfully, not that such execution is possible. For such guarantees, we must augment the current execution module with some lookahead ability, as Nau et al. (1999) have already done for hierarchical task networks. This will require additional effort from the agent, but still far less than solving a problem with means-ends analysis.

Another response would use inductive logic programming or related methods to learn additional conditions on skill clauses that ensure they will achieve their goal, even without lookahead. To this end, we can transform the results of lookahead search into positive and negative instances of clauses, based on whether they would lead to success, much as in early work on inducing search-control rules from solution paths (Sleeman et al., 1982). Even if such conditions are incomplete, they should still reduce the planning effort required to ensure the agent's actions will produce the desired outcome.

Another important limitation concerns our assumption that the agent always executes a skill to achieve a desired situation. The ability to express less goal-directed activities, such as playing a piano piece, are precisely what distinguishes hierarchical task networks from classical planning (Erol, Hendler, & Nau, 1994). We hope to extend our framework in this direction by generalizing its notion of goals to include concepts that describe sets of situations that hold during certain time intervals. To support the hierarchical skill acquisition, this augmented representation will require extensions to both the problem solving and learning mechanisms. In addition, we should extend our framework to handle skill learning in nonserializable domains, such as tile-sliding puzzles, which motivated much of the early research on macro-operator formation (e.g., Iba, 1988).

Future work should also address a related form of overgeneralization we have observed on the Tower of Hanoi puzzle. In this domain, the approach learns reasonable hierarchical skills that can solve the task without problem solving, but that only do so about half the time. In other runs, the learned skills attempt to move the smallest disk to the wrong peg, which ultimately causes the system to fail. Humans often make similar errors but also learn to avoid them with experience. Inspection of the behavioral trace suggests this happens because one learned skill clause includes variables that are not mentioned in the head but are bound in the body. We believe that including contextual conditions about variables bound higher in the skill hierarchy will remove this nondeterminism and produce more correct behavior.

In addition, recall that the current system does not chain backward from the start condition of learned skill clauses. We believe that cases will arise in which such chaining, even if not strictly necessary, will make the acquisition of complex skills much easier. Extending the problem solver to support this ability means defining new conceptual predicates that the agent can use to characterize situations in which its learned skills are applicable. This will be straightforward for some domains and tasks, but some recursive skills will need recursively defined start concepts, which requires a new learning mechanism. Augmenting the system in this manner may also lead to a utility problem (Minton, 1990), not during execution of learned teleoreactive logic programs but during the problem solving used for their acquisition, which we would then need to overcome.

Finally, 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 other objects involved, but it cannot learn a program for constructing a tower with arbitrarily specified components. Extending the system's ability to transfer across different goals, including ones that are defined recursively, is another important direction for future research on learning hierarchical skills.

8. 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 use 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 – acquires recursive, executable skills in an incremental and cumulative manner. We reported experiments that demonstrated the method's ability to acquire hierarchical and recursive skills for three domains, along with its capacity to transfer its learned structures to tasks with more objects than seen during training.

Teleoreactive logic programs incorporate ideas from a number of traditions, including logic programming, adaptive control, and hierarchical task networks, in a manner that supports reactive but goal-directed behavior. The approach which we have described for acquiring such programs, and which we have incorporated into the ICARUS architecture, borrows intuitions from earlier work on learning through problem solving, but its details rely on a new mechanism that bears little resemblance to previous techniques. Our work on learning teleoreactive logic programs is still in its early stages, but it appears to provide a novel and promising path to the acquisition of effective control systems through a combination of reasoning and experience.

Acknowledgements

This material is based on research sponsored by DARPA under agreement numbers HR0011-04-1-0008 and FA8750-05-2-0283 and by Grant IIS-0335353 from the National Science Foundation. The U. S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. Discussions with Nima Asgharbeygi, Kirstin Cummings, Glenn Iba, Negin Nejati, David Nicholas, Seth Rogers, and Stephanie Sage contributed to the ideas we have presented in this paper.

References

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

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

Choi, D., Kaufman, M., Langley, P., Nejati, N., and Shapiro, D. 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, 2004.

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

Clocksin, W. F., and Mellish, C. S. *Programming in* PROLOG. Berlin: Springer-Verlag, 1981.

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

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

Fern, A., Yoon, S. W., and Givan, R. 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, 2004.

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

Jones, R. M., and Langley, P. A constrained architecture for learning and problem solving. *Computational Intelligence*, *21*, 480–502, 2005.

Khardon, R. Learning action strategies for planning domains. *Artificial Intelligence*, *113*, 125–148, 1999.

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

Langley, P., and Rogers, S. Cumulative learning of hierarchical skills. *Proceedings of the Third International Conference on Development and Learning*. San Diego, CA, 2004.

Marsella, S., and Schmidt, C. F. A method for biasing the learning of nonterminal reduction rules. In S. Minton (Ed.), *Machine learning methods for planning*. San Mateo, CA: Morgan Kaufmann, 1993.

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

Mooney, R. J. 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, 1989.

Nau, D., Cao, Y., Lotem, A., and Muñoz-Avila, H. SHOP: Simple hierarchical ordered planner. *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence*, pp. 968–973. Stockholm: Morgan Kaufmann, 1999.

Neches, R., Langley, P., and Klahr, D. Learning, development, and production systems. In D. Klahr, P. Langley, and R. Neches (Eds.), *Production system models of learning and development*. Cambridge, MA: MIT Press, 1987.

Newell, A., and Simon, H. A. GPS, A program that simulates human thought. In H. Billing (Ed.), *Lernende automaten*. Munich: Oldenbourg KG. Reprinted in E. A. Feigenbaum & J. Feldman (Eds.), *Computers and thought*. New York: McGraw-Hill, 1961.

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

Olsson, R. Inductive functional programming using incremental program transformation. *Artificial Intelligence*, *74*, 55–83, 1995.

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

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

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

Schmid, U. A cognitive model of learning by doing. *Models and human reasoning – Festschrift für Bernd Mahr*. Berlin: Wissenschaft & Technik Verlag, 2005.

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

Sleeman, D., Langley, P., and Mitchell, T. Learning from solution paths: An approach to the credit assignment problem. *AI Magazine*, *3,* 48–52, 1982.

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

Veloso, M. M., and Carbonell, J. G. Derivational analogy in PRODIGY: Automating case acquisition, storage, and utilization. *Machine Learning*, *10,* 249–278, 1993.

Zelle, J. M., and Mooney, R. J. 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, 1993.