Interleaving Learning, Problem Solving, and Execution in the Icarus Architecture

Pat Langley (langley@csli.stanford.edu)
Dongkyu Choi (dongkyuc@stanford.edu)
Seth Rogers (srogers@csli.stanford.edu)

Computational Learning Laboratory
Center for the Study of Language and Information
Stanford University, Stanford, CA 94305 USA

Abstract

In this paper, we review Icarus, a cognitive architecture that utilizes hierarchical skills and concepts for reactive execution in physical environments. In addition, we present two extensions to the framework. The first involves the incorporation of means-ends analysis, which lets the system compose known skills to solve novel problems. The second involves the storage of new skills that are based on successful means-ends traces. We report experimental studies of these mechanisms on three distinct domains. Our results suggest that the two methods interact to acquire useful skill hierarchies that generalize well and that reduce the effort required to handle new tasks. We conclude with a discussion of related work on learning and prospects for additional research.

1. Introduction and Motivation

Research on cognitive architectures (Newell, 1990) attempts to understand the computational infrastructures that support intelligent behavior. A specific architecture characterizes the aspects of a cognitive agent that remain the same across time and over different domains, and typically makes strong commitments about the representation of knowledge structures and the processes that operate on them. Learning has been a central concern in most architectural research, with a variety of mechanisms having been proposed to model the acquisition of knowledge from experience. The learning methods embedded in most cognitive architectures are incremental, reflecting evidence that humans acquire knowledge in this manner, but there have been few accounts of the origin of hierarchical structures that appear crucial to complex cognition.

In this paper we review ICARUS, a candidate architecture that diverges from its predecessors on a number of dimensions. One important difference is that typical architectures handle conceptual knowledge in a procedural manner, typically using production rules, whereas our framework contains separate memories for concepts and skills. Another distinctive feature is that most architectures are based on production systems, which encode knowledge as a 'flat' set of condition-action rules, whereas ICARUS makes an architectural commitment to the hierarchical organization of knowledge. One can certainly encode hierarchical structures in frameworks like ACT-R (Anderson, 1993) and Soar (Laird, Rosenbloom, & Newell, 1986), but this remains the modeler's choice rather than a strong theoretical claim. In addition, most cognitive architectures evolved from theories of human problem solving, which has led to subordinate roles for perception and action even in those frameworks that support them.[1] In contrast, ICARUS is primarily an execution architecture that perceives and reacts to external environments, which we view as more basic than problem solving.

However, ICARUS' reliance on hierarchical structures raises key questions about their origin. Moreover, the architecture's emphasis on execution does not mean that mental activities like problem solving are unimportant, since they can let an agent handle novel tasks for which stored knowledge is unavailable. The central hypothesis of this paper is that hierarchical skills arise, at least in many cases, from problem-solving behavior, and that, once learned, the agent can use these structures to support reactive execution in the environment. Moreover, this acquisition occurs in an incremental manner, with new skills being learned gradually as the agent encounters new problems it cannot handle without resorting to problem solving.

We refer to ICARUS as a 'cognitive architecture' in the same sense that the Soar community uses that expression. Both frameworks aim for consistency with general knowledge about human cognition and hope to support the same broad range of abilities that people demonstrate. However, our current research does not attempt to match at a fine-grained level the results of psychological experiments, as done with architectures like ACT-R. We may address such issues in future research, but for now we are concerned with coarse regularities that demand explanation, such as the apparent hierarchical nature of human skills and their incremental acquisition from experience.

In the sections that follow, we review ICARUS' representation and organization of concepts and skills, along with the inference and execution processes that utilize them. After this, we present

---

1. Recent extensions to Soar and ACT-R have provided them with sensori-motor interfaces, but their emphasis on central cognition remains strong.

a new module that interleaves means-ends problem solving with execution when known skills are insufficient to solve a task. Next we describe a mechanism for creating generalized skills from traces of successful problem solving that supports incremental, hierarchical learning. We report experiments with this learning mechanism that demonstrate its ability to generalize to novel situations and reduce effort on new problems. In closing, we discuss earlier research on learning for problem solving and execution, along with some directions for future work.

2. Representation and Organization

Like other cognitive architectures, ICARUS makes commitments to its representation of knowledge, the manner in which that knowledge is organized, and the memories in which it resides. Following most theories of human cognition, the framework distinguishes between long-term memories, which change only gradually due to learning, and short-term memories, which change rapidly as the agent revises its beliefs and goals. In this section, we discuss ICARUS' memories and the formalisms used to encode their contents.[2] We will take our examples from the Blocks World, since many readers should find this domain familiar. We have described these aspects of the framework in more detail elsewhere, including their use in other domains like in-city driving (Choi et al., 2004) and multi-column subtraction (Langley, Cummings, & Shapiro, 2004).

2.1 Long-Term Conceptual Memory

One of ICARUS' long-term memories stores concepts that describe generalized situations in the environment. These may involve isolated objects, such as individual blocks, but they can also characterize physical relations among objects, such as the relative positions of blocks. Long-term conceptual memory contains the definitions of these logical categories. Each element specifies the concept's name and arguments, along with fields which describe perceptual entities that must be present, lower-level concepts that must match, lower-level concepts that must not match, and numeric relations that must be satisfied. Table 1 presents some concepts from the Blocks World. For example, the relation *on* describes a perceived situation in which two blocks have the same x position and the bottom of one has the same y position as the top of the other. The concept *clear* instead refers to a single block, but one that cannot hold the relation *on* to any other.

Definitions of this sort organize ICARUS categories into a conceptual hierarchy. Primitive concepts are defined entirely in terms of perceptual conditions and numeric tests, but many incorporate other concepts in their definitions. This imposes a lattice structure on the memory, with more basic concepts at the bottom and more complex concepts at higher levels. The resulting hierarchy is similar in spirit to discrimination network models of human memory like EPAM (Richman, Staszewski, & Simon, 1995), as well as to frameworks like description logics (Nardi & Brachman, 2002). Structurally, this lattice bears a close resemblance to the Rete networks (Forgy, 1982) used for matching in production-system architectures.

---

2. Previous versions of ICARUS, reported by Langley et al. (1991) and by Shapiro et al. (2001), have made substantially different assumptions. To distinguish the current architecture from its predecessors, ICARUS/3 would be a more proper reference.

Table 1. Some Icarus concepts for the Blocks World, with variables indicated by question marks. Percepts refer only to attribute values used elsewhere in the concept definition.

```
(on (?block1 ?block2)
 :percepts  ((block ?block1 xpos ?xpos1 ypos ?ypos1)
             (block ?block2 xpos ?xpos2 ypos ?ypos2 height ?height2))
 :tests     ((equal ?xpos1 ?xpos2)
             (>= ?ypos1 ?ypos2)
             (<= ?ypos1 (+ ?ypos2 ?height2)))
(clear (?block)
 :percepts  ((block ?block))
 :negatives ((on ?other ?block))
(unstackable (?block ?from)
 :percepts  ((block ?block) (block ?from))
 :positives ((on ?block ?from) (clear ?block) (hand-empty)))
(pickupable (?block ?from)
 :percepts  ((block ?block) (table ?from))
 :positives ((ontable ?block ?from) (clear ?block) (hand-empty)))
```

2.2 Long-Term Skill Memory

Icarus also incorporates a second long-term memory that stores knowledge about skills it can execute in the environment, including their conditions for application and their expected effects. Each skill clause includes a head (a name and zero or more arguments) and a body that specifies the concepts that must hold to initiate the skill and one or more components. A primitive skill clause indicates one or more ordered, executable actions, along with those concepts that, taken together, describe the situation the skill produces when done. A primitive skill may also state conditions that must hold throughout its execution, which may require multiple cycles to complete. For example, Table 2 shows the skill *pickup*, which must satisfy the start condition, *(pickupable ?block ?from)*, defined in Table 1, and invokes *∗grasp*, which grasps a block, and *∗vertical-move*, which moves the hand in the vertical direction. The skill's only stated effect is to make *(holding ?block)* true.

In contrast, a nonprimitive skill clause specifies how to decompose that activity further. For instance, Table 3 includes two clauses for the nonprimitive skill *clear*. Each indicates that executing the clause will achieve that goal, but they differ in their start conditions and in their subskills. Nonprimitive skill clauses do not specify either required conditions or effects, but their heads always corresponds to a concept that the skill will achieve upon successful completion. This representational assumption figures centrally in the learning mechanism we describe later. Because Icarus concepts and skills utilize a syntax similar to that found in the programming language Prolog, we have referred elsewhere to sets of these long-term memory structures as *teleoreactive logic programs* (Choi & Langley, 2005). This phrase conveys both their structural similarity to traditional logic programs and their ability to behave reactively in a goal-driven manner, following Nilsson's (1994) notion of a teleoreactive system.

Table 2. Primitive skills for the Blocks World. Each clause has a head that specifies the skill's name and arguments, a set of typed percepts, a single start condition, a set of effects, and a set of executable actions (marked by asterisks).

```
(unstack (?block ?from)
 :percepts  ((block ?block) (block ?from))
 :start     ((unstackable ?block ?from))
 :effects   ((clear ?from) (holding ?block))
 :actions   ((*grasp ?block) (*vertical-move ?block)))
(pickup (?block ?from)
 :percepts  ((block ?block) (table ?from))
 :start     ((pickupable ?block ?from))
 :effects   ((holding ?block))
 :actions   ((*grasp ?block) (*vertical-move ?block)))
(stack (?block ?to)
 :percepts  ((block ?block) (block ?to))
 :start     ((stackable ?block ?to))
 :effects   ((on ?block ?to) (hand-empty))
 :actions   ((*horizontal-move ?block ?xpos)
             (*vertical-move ?block)
             (*ungrasp ?block)))
(putdown (?block ?to)
 :percepts  ((block ?block) (table ?to))
 :start     ((putdownable ?block ?to))
 :effects   ((ontable ?block ?to) (hand-empty))
 :actions   ((*horizontal-move ?block)
             (*vertical-move ?block)
             (*ungrasp ?block)))
```

2.3 Short-Term Memories

In addition to long-term memories, which encode relatively stable knowledge about a domain, ICARUS follows standard psychological theory by incorporating short-term stores that change more rapidly. These contain the agent's temporary perceptions and beliefs about the environment, as well as its goals and intended activities. They include:

- a *perceptual buffer* that holds descriptions of physical entities which correspond to the output of sensors; for the blocks world, this includes literals like *(block B xpos 10 ypos 2 width 2 height 2)*, which specify the position and size of individual blocks.

- a *short-term conceptual memory* that contains beliefs about the environment which the agent infers from items present in its perceptual buffer and long-term concept memory; for instance, this might contain the instance *(on B C)*, which is an instance of the *on* concept in Table 1.

- a *short-term skill memory* that contains the agent's goals and associated skill instances it intends to execute; each goal literal specifies a concept's name and argumets, as in *(clear A)*, whereas each associated intention gives a skill's name and its arguments, as in *(stack B C)*, which is an instance of the skill *stack* in Table 2.

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

```
(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)))
```

Unlike most cognitive architectures, every element in the short-term conceptual and skill memories must be an instance of some generalized structure in the long-term conceptual and skill memory, respectively; they cannot be arbitrary symbolic structures. We have discussed this strong correspondence assumption at more length elsewhere (Langley & Rogers, 2005).

3. Conceptual Inference and Skill Execution

Like most cognitive architectures, ICARUS operates in distinct cycles. On each such iteration, the system updates its perceptual buffer by sensing objects in its field of view, with the specific sensors depending on the particular environment in which the agent is operating. This process produces perceptual elements, which are are deposited in the perceptual buffer and which initiate matching against long-term concepts. The matcher checks to see which primitive concepts (i.e., those defined entirely in terms of percepts) are satisfied, adds each matched instance to conceptual short-term memory, and repeats the process on nonprimitive concepts to infer higher-level beliefs.

In this way, ICARUS infers all instances of concepts that are implied by its conceptual definitions and the contents of the perceptual buffer. For example, a Blocks World agent would first update its descriptions of the blocks and the table, then infer primitive concepts like *on*, and finally infer complex concepts like *unstackable*. This bottom-up procedure operates in much the same way as the Rete networks (Forgy, 1982) used in many production-system architectures and the logical inference methods used in many truth-maintenance systems (e.g., Doyle, 1979). The default process is exhaustive, but elsewhere we have reported an alternative mechanism that makes inferences more selectively (Asgharbeygi et al., 2005).

On each cycle, the architecture also examines the agent's goals and their associated intentions in short-term skill memory to determine which, if any, apply to the current situation.[3] For each intended skill instance, ICARUS accesses all clauses of the general skill to see if they are applicable.

---

3. ICARUS' first step in a run typically involves selecting a relevant and applicable instance of a nonprimitive skill that it believes will achieve one of its goals.

Since variables can be bound within a skill's body, this set may include multiple variants of each skill clause stored in long-term memory. A primitive skill clause is applicable if, for its current variable bindings, its effects do not yet hold, its requirements are satisfied, and, if the system has not yet started executing it, the start conditions match the current situation. A higher-level skill clause is applicable if its head is not satisfied, the start conditions are satisfied if it has not been initiated, and at least one subskill is applicable. Because this latter test is recursive, a skill is applicable only when ICARUS can find at least one acceptable path downward to executable actions, which the architecture returns for invocation.

For example, suppose an ICARUS agent has the goal *(clear A)* 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. Suppose further that the agent has access to the primitive skills in Table 2 and the nonprimitive ones in Table 3. In this case, the system would find an applicable path through the skill hierarchy that is relevant to its goal: [*(clear A)*, *(unstackable B A)*, *(clear B)*, *(unstackable C B)*, *(clear C)*, *(unstack C B)*]. This holds because the instantiated start conditions of each skill along the path (e.g., *(on B A)* and *(hand-empty)* for the topmost skill) are present in conceptual short-term memory. If selected, *(unstack C B)* would alter the environment, making the path [*(clear A)*, *(unstackable B A)*, *(clear B)*, *(unstackable C B)*, *(hand-empty)*, *(putdown C T)*] acceptable on the next cycle. This would produce a belief state that enables the next step in the procedure, which would continue until the agent had satisfied its top-level goal, *(clear A)*.

During skill selection, ICARUS incorporates two preferences that provide a balance between reactivity and persistence. When confronted with a choice between two or more subskills, it selects the first alternative for which the head is not satisfied. This supports reactive control, since the system reconsiders previously completed subskills and, if their effects no longer hold for some reason, reexecutes them to remedy the problem. On the other hand, when encountering two or more applicable skill paths, ICARUS selects the one that shares the most elements from the start of the path executed on the previous cycle. This encourages the system to continuing executing a high-level skill it has already started until that skill achieves its associated goal or until it becomes inapplicable.

4. Means-Ends Problem Solving

As just explained, ICARUS can execute complex hierarchical skills in a reactive manner, but our initial studies (e.g., Choi et al., 2004; Langley et al., 2004) assumed that these skills are already present in long-term memory. Although much human behavior appears to involve the application of such routine skills, people can also solve novel tasks that require the dynamic combination of existing knowledge elements through some form of heuristic problem solving.

To model this capability in ICARUS, we have introduced a variant of means-ends analysis (Newell, Shaw, & Simon, 1960) that operates over the architecture's knowledge structures, including both long-term concepts and skills provided by the programmer and short-term beliefs and goals produced by the architecture. Traditional means-ends problem solving selects some unsatisfied aspect of the goal state to achieve, then selects an operator that would achieve it. If that operator's preconditions match the current state, it is applied; otherwise, the method selects an unsatisfied

precondition to achieve, selects an operator that would achieve it, and so on. Once a condition is met, the process is repeated until the original goal description is satisfied. This may require search, which is often pursued in a depth-first manner. Means-ends analysis has been implicated repeatedly in human problem solving on novel tasks.

To support this mechanism, our extended version of ICARUS augments the short-term skill memory with a goal stack. Each element in this stack specifies a goal (a desired concept instance), whether the agent intends to achieve it by backward chaining off a concept definition or a skill clause, and, in the latter case, the skill instance that, if executed, should achieve it. Each goal element also specifies subgoals that have already been achieved, along with skill and/or concept instances that it has tried in reaching this goal but that have failed. The first are needed to keep the system from considering skills that would undo its previous accomplishments, whereas the second ensures it does not repeat earlier mistakes. We also assume that both the start conditions of primitive skills and top-level goals must be cast as single relational literals, which causes no loss in generality, since either may be defined concepts.

We have also extended the ICARUS interpreter to take advantage of these new memory structures. On each cycle, the system takes one of five distinct, ordered steps:

1. If the current goal G of the goal stack GS is satisfied, then pop G from GS and store information about the success with G's parent.

2. If the goal stack GS does not exceed the depth limit and there are applicable skill paths that start from a skill instance with the current goal G as its head, then select one such path and execute it.

3. If there is a nonempty set of primitive skill instances in which the current goal G is an effect that have not already failed, then select a skill instance from this set and push its start condition (which we assume subsumes any required conditions) onto the goal stack GS.

4. If the current goal G is an instance of a complex concept with unsatisfied subconcepts H and with satisfied subconcepts F, then if there is a subconcept I in H that has not yet failed, push I onto the goal stack GS.

5. Otherwise pop the current goal G from the goal stack GS and store information about the failure with G's parent.

We assume that each of these activities takes a single cycle of the architecture, with the initial situation being a special case of the third item that triggers the process. Because reasoning about how to achieve an objective can require many manipulations of the goal stack, it takes more cycles than executing a stored hierarchical skill for that goal, even when the agent finds a solution on its first attempt and does not have to backtrack.

Figure 1 shows a successful trace of the problem solver's behavior on a simple Blocks World task in which when the goal is *(clear A)* and when block A is on the table, block B is on A, block C is on B, and the hand is empty. In this situation, the system looks for executable skills with this goal as its head but, when this fails, it considers skills that have the goal as one of its effects. In this case, invoking the primitive skill instance *(unstack B A)* would produce the intended result, but it
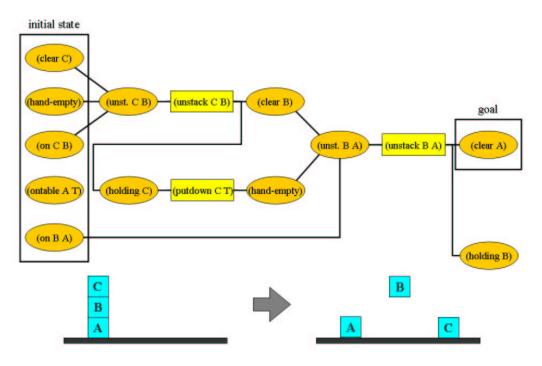
Figure 1. A trace of successful problem solving in the Blocks World, which ellipses indicating goals and rectangles denoting primitive skills.

cannot be applied because its instantiated start condition, *(unstackable B A)*, does not hold. In response, the problem solver stores the skill instance with the initial goal and pushes the subgoal onto the goal stack.

Next, the problem solver attempts to retrieve skills that would achieve *(unstackable B A)*. However, 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 system pushes it onto the goal stack. This in turn leads it to consider 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 system executes *(unstack C B)*. The associated actions alter the environment and cause 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)*. However, 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.

For the sake of clarity, both our description and Figure 1 present the trace of successful problem solving, but finding such a solution may involve search. When backward chaining off skills that would achieve the objective of the current stack entry, ICARUS considers only skill instances that have not yet failed. The system also prefers candidates that have the fewest expanded start conditions that are unmet by the current environmental state, with fully matched conditions being most desirable. If candidates tie on this criterion, it selects an alternative at random. When backward chaining off the unmatched elements of a concept definition, the system selects subgoals at random after eliminating those which have failed in the past.

Taken together, these biases produce a heuristic version of means-ends analysis. However, this problem-solving method is tightly integrated with the execution process. ICARUS backward chains off concept or skill definitions when necessary, but it executes the skill associated with the top stack entry as soon as it becomes applicable. Moreover, because the architecture can chain over hierarchical reactive skills, their execution may continue for many cycles before problem solving is resumed. In contrast, most models of human problem solving and most AI planning systems focus on the generation or the execution of plans, rather than interleaving the two processes.

Of course, executing a component skill before constructing a complete plan can lead an agent into difficulties, since one cannot always backtrack in the physical world. This strategy may well lead to suboptimal behaviors, but human intelligence is more about satisficing than optimizing, and interleaving problem solving with execution requires far less memory than constructing a full plan before executing it. However, it can produce situations from which the agent cannot recover without starting the problem over.

In such cases, ICARUS stores the goal element for which its executed skill caused a problem, along with everything below it in the stack. The system begins the problem again, this time avoiding the skill and selecting another option. If it makes a different execution error this time, it again stores the problematic skill and its context, then starts over once more. ICARUS also starts over if it has not achieved the top-level objective within a specified number cycles. Such repeated attempts at solving a task, with selected memory about previous passes, seems a better model of human problem solving than systems that construct a complete plan before execution. Jones and Langley's (in press) model of means-ends problem solving, EUREKA, used a similar restart strategy, but it kept no explicit record of previous failed paths.

5. Learning Hierarchical Skills from Problem Solving

In the previous pages, we described two facets of ICARUS: its execution of hierarchical skills on familiar tasks and its use of problem solving to handle novel ones. The first lets the system operate efficiently, but skills are tedious to construct manually, whereas the second gives the system flexibility but requires reasoning and means-ends search. We believe that humans also have both capabilities, but that they use learning to transform the results of successful problem solving into hierarchical skills. We would like to incorporate a similar capability into ICARUS.

However, we want our learning mechanisms to reflect certain properties that appear to hold for human skill acquisition. One is that learning should take advantage of existing knowledge, such as

the definitions of current skills and concepts. In addition, acquisition should be incremental, in that it learns from each new experience, and interleaved with the problem-solving process. The recent literature on computational learning contains few cases of such knowledge acquisition, although in Section 7 we discuss some older work that has this character.

Our extension of ICARUS achieves this effect through a form of impasse-driven learning that is tied closely to its problem-solving and execution processes. For this reason, the learning mechanisms require no additional inputs beyond those required for these basic performance processes. As in SOAR (Laird et al., 1986), the purpose of skill learning is to avoid such impasses in the future. Thus, whenever the architecture achieves an objective that is associated with an entry in the goal stack, this success provides an opportunity for learning. The system acquires two distinct forms of skill that are tied to different aspects of problem solving.

The first class of skills result from situations in which the problem solver cannot find a skill to achieve a goal G, and thus pursues subgoals based on the unsatisfied conditions of G's conceptual definition. If the agent achieves these subgoals in the order $\{G_1, G_2, \ldots, G_n\}$, thus satisfying the parent goal G, ICARUS constructs a new skill clause that has G as its head and that has $\{G_1, G_2, \ldots, G_n\}$ as its ordered subskills.[4] The start conditions of the new clause are simply those subconcepts of G that were satisfied when it was pushed onto the goal stack. The head, conditions, and subskills have their arguments replaced by variables in a consistent manner, ensuring applicability to analogous situations that involve different objects.

For example, upon achieving the subgoal *(unstackable B A)* in Figure 1, the system constructs the *unstackable* skill clause labeled 3 in Table 3. The head *(unstackable ?B ?A)* is a generalized version of the goal *(unstackable B A)*, whereas the ordered subskills *(clear ?B)* and *(hand-empty)* are generalized versions of its two subgoals *(clear B)* and *(hand-empty)*. The start conditions are *(on ?B ?A)* and *(hand-empty)*, which are generalized versions of the subconcepts that held when the goal was established. Finally, the `:percepts` field specifies the types for objects that serve as the head's arguments. This mechanism constructs different variants of a skill, with separate start conditions and distinct subskills, from subproblems that involve different initial conditions.

The second category results from situations in which ICARUS has selected a primitive skill instance $S2$ in order to achieve a goal G, but found its single start condition $G2$ unsatisfied and selected another skill instance, $S1$, to achieve it. Once the agent has executed both skills successfully and it has reached the goal, the system constructs a new skill clause that has G as its head and that has $G2$ (rather than the specific clause $S1$) and $S2$ as ordered subskills. The start conditions are simply the start conditions of the $S1$ clause used in the subproblem solution, which are sufficient because the problem solver $S1$ selected it to achieves the start condition of $S2$, which in turn achieves the goal G. Again, specific arguments are replaced consistently by variables.

For instance, upon achieving the top-level goal *(clear A)* in Figure 1, ICARUS creates the *clear* skill clause labeled 4 in Table 3. This incorporates a generalized version of *(clear A)* as its head, along with variableized versions of *(unstackable B A)* and *(unstack B A)* as its two ordered subskills. The start conditions, *(on ?B ?A)* and *(hand-empty)*, are the same as those for *unstackable* clause

---

4. Note that the system refers to subskills by the goals they achieve, rather than to specific clauses, which lets the parent skill take advantage of other clauses for these goals that are learned later.

3 just discussed, since the latter was created to achieve the start condition of *unstack* under those same conditions, which in turn satisfies the goal *clear*.

Both learning mechanisms are fully incremental, in that each learning event draws on a single problem-solving experience and thus requires no memory of previous ones. They support within-trial learning, since skills acquired on one subproblem may be used to handle later subproblems. The processes also build on existing knowledge, since the construction of new skill clauses involves the composition of those used in a training problem's solution. Taken together, these support a form of cumulative learning, in which ICARUS learns skills on one problem, uses them to solve a later problem, and incorporates them into still higher-level structures.

As suggested by our examples, these learning methods can acquire both disjunctive and recursive skills. The key to this ability lies in the assumption that acquired skill clauses which achieve the same goal should be given the same head. By indexing skills in this manner, ICARUS knows when two or more clauses should be stored together, which leads in turn to the creation of skills that call on themselves, either directly or through intermediate skills. This makes the architecture's learned skills considerably more flexible and general than traditional 'macro-operators' (e.g., Iba, 1988) or composed production rules (e.g., Neves & Anderson, 1981).

Of course, the creation of disjunctive and recursive structures has potential for overgeneralization, as demonstrated by research on the induction of context-free grammars (e.g., Langley & Stromsten, 2000). Our technique for determining the start conditions on new skill clauses is much simpler than standard techniques for analytical learning or rule induction. In fact, at first glance, the learned clauses in Table 3 appear highly overgeneral, but this ignores the fact that ICARUS does not interpret skills in isolation. Recall that the architecture must find an entire path through the skill hierarchy before it can execute the primitive skill at its terminus. This means the system collects conditions dynamically, as it descends the hierarchy, guarding against overgeneralization by carrying out limited analysis at performance time rather than doing it all at learning time.

Unlike some approaches to incremental learning, ICARUS' methods require no additional mechanisms for skill refinement. Each skill clause is generalized when the architecture constructs it, and its start conditions are assumed to be accurate. The knowledge it acquires from solving a given problem may well be incomplete, but this will simply lead to further impasses that produce additional learning. Skill clauses acquired later complement, but do not compete with, those learned earlier because they cover different situations or the older clauses would have avoided the impasse. Thus, learning is purely monotonic, as in frameworks like Soar.

We should note that our current implementation restricts the use of learned skills in future problem solving. In particular, we have adopted Mooney's (1989) idea that one should not chain off the preconditions of learned skills. This does not restrict their use by the execution module, but it does mean that the problem solver considers a learned skill only when its start conditions are already satisfied. As a result, clauses acquired from chaining off skills always have a left-branching structure in which the second subskill is primitive. This assumption may seem restrictive, but, like Mooney, we believe it provides an effective guard against the utility problem (Minton, 1990), in which the creation and use of complex structures reduces search but actually slows performance.

6. Experimental Studies of Skill Learning

Although the new methods for learning hierarchical skills seem plausible, whether they improve an ICARUS agent's performance is an empirical question. In this section, we report the results of basic tests of these mechanisms on three distinct domains: in-city driving, the Blocks World, and FreeCell solitaire. After this, we report more systematic experiments with the domains that examine the effects of learning in more detail. As one measure of performance, we used the number of recognize-act cycles required to solve the problem in the simulated environment, including both problem solving and execution steps. However, we also measured the CPU time required to solve each problem, to determine whether ICARUS suffers from the utility problem.

6.1 Domains and Basic Demonstrations

To ensure that our approach to learning hierarchical skills operated as intended, we developed ICARUS programs for the three domains. In each case, we provided a set of primitive skills sufficient for solving problems with means-ends analysis and a set of hierarchical concepts sufficient for recognizing situations that were relevant to executing those skills. For example, we devised some 41 concepts and 19 skills for the in-city driving domain, 11 concepts and four skills for the Blocks World, and 24 concepts and 12 skills for FreeCell solitaire. The Appendix gives the names of the primitive concepts, nonprimitive concepts, and primitive skills provided for each domain, which should also suggest their function. In addition, we also provided the architecture with a set of sensors and effectors for each simulated environment.

We have already discussed the Blocks World, but both in-city driving and FreeCell merit some explanation. The first domain involves a dynamic simulation of a downtown driving environment. The city contains objects represented as rectangles of different sizes, including buildings and sidewalks organized into square blocks that are divided by street segments and intersections. Each segment includes a yellow center line and white dotted lane lines, and it has a marked street name and speed limit. Each buildings has a unique street address to help the agent navigate through the city and to support tasks like package delivery. The city configuration used in our experiments has nine blocks with four vertical streets and four horizontal streets. The ICARUS agent must operate under physical laws and follow the rules of driving, such as staying on the right side of the street and turning from the proper lane. We provide the agent specific with goals to achieve, such as getting onto another street segment or delivering a package to a certain address.

FreeCell solitaire involves eight stacks of cards, the first four of which contain seven cards and the last four contain six cards. All 52 cards are dealt face up, making them visible to the player. In addition, there are four free cells, which can serve as temporary holding spots for one card each during the game, and one foundation cell for each suit. The goal in FreeCell is to get all cards on the foundation cells in ascending order (where the ace is one and the king is thirteen) grouped by suit. Once on its foundation cell, a card cannot be removed. Only fully-exposed cards at the top of each stack and cards that in the free cells are in play. The agent can move one card at a time to an available free cell, to the appropriate foundation cell, to an empty stack, or to a stack in which the top card has a different color and value one higher than the moved card.

Sample runs with the in-city driving domain, the Blocks World, and a reduced version of FreeCell indicated that the extended version of ICARUS was able to solve problems in their respective domains with some search and, from their respective traces, learn hierarchical skills in the manner described earlier. We found that, when given the same task to solve a second time, the system utilized this knowledge to handle it without problem solving. Moreover, because the system generalizes its learned structures beyond the specific instances on which they are based, they transfer fully to any tasks that are isomorphic to those it has already solved. The only constraint is that this isomorphism must involve the same goal and have the same concepts satisfied or unsatisfied in the initial environment.

However, we should note this ability does not mean that the system can complete a familiar problem in a single cycle. Recall that, traditional work on cognitive architectures, ICARUS resorts to problem solving only to enable action, and it must still execute its acquired skills to achieve a goal. Thus, for a problem that requires four primitive steps, the system takes six cycles on the second encounter, with one to retrieve the hierarchical skill and one to realize it has finished. However, the agent requires neither search or backward chaining over skills or concepts to complete any problem it has solved previously.

6.2 Experiment with In-City Driving

Although these initial runs were encouraging, we desired more than anecdotal demonstrations that the new mechanisms supported incremental learning of hierarchical skills. We also wanted evidence from systematic experiments that this learned knowledge produces more effective behavior. Our first study along these lines focused on in-city driving, which is the most dynamic of the three settings and thus the one most appropriate for evaluating our methods for learning skills that support reactive execution.

As noted above, we provided ICARUS with 41 concepts and 19 primitive skills relevant to this environment. With the basic knowledge, the agent can characterize its 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 this is not sufficient to drive safely in a city environment. The agent must still learn skills for staying aligned and centered within lane lines, change lanes, increase or decrease speed for turns, and stop for parking.

To encourage such learning, we presented the agent with the goal of driving on a different street segment than its current one. To achieve this objective, 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 into the target segment, turning the corner, and finally aligning and centering in the new lane. We let the ICARUS agent practice this task for five trials to examine its improvement with experience. We repeated this procedure ten different times with slightly different starting positions, collected performance measures for each run, and averaged the results.

Figure 2 shows the total number of cycles as a function of the number of trials, along with the number of planning and execution cycles required to achieve the goal. As the agent accumulates knowledge about this task, problem solving disappears almost entirely, which causes the reduced
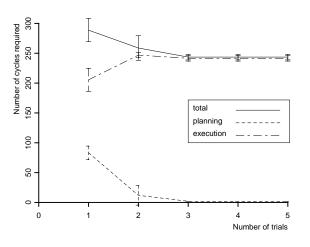
Figure 2. The total number of cycles required to solve a particular right-turn task along with the planning and execution times, as a function of the number of trials. Each learning curve shows the mean over ten sets of trials and 95 percent confidence intervals.

number of total cycles. However, this problem is dominated by execution time, since the agent must actually drive the vehicle to its destination. Execution cycles appear to increase, which occurs not because the learned skills are inefficient but rather because time progresses even during problem solving. Thus, the vehicle moves in the right direction during this period in the early trials, reducing the distance that remains to travel. As problem solving becomes unnecessary, the agent drives this extra distance under conscious control rather than accidentally. CPU time remained approximately the same with increased experience, presumably for the same reasons.

Table 4 shows the five skill clauses acquired during one of these runs. The two clauses for *driving-in-segment* specify different decompositions for achieving this top-level goal under alternative start conditions. The second of these refers to the clause for *in-segment*, which refers to the learned sub-skill for *in-intersection-for-right-turn* and the primitive skill *steer-for-right-turn*. The former refers to *in-rightmost-lane*, which invokes the primitive skill clause *driving-in-segment*, but it also calls on itself recursively with distinct arguments. For clarification, the table also presents the primitive clause for *in-intersection-for-right-turn*, which the system was given as background knowledge.

Figure 3 shows a trace of the agent's behavior on the task during learning, in a situation that involves a street with two lanes, and afterwards, in a setting that instead involves three lanes. The trace of the vehicle's movement demonstrates that the learned skills generalize to cases that involve more lanes than were present during training. This ability follows directly from the recursive structure of the learned *in-intersection-for-right-turn* clause. Behavior after learning is also smoother, presumably because the agent need not engage in problem solving when it overshoots slightly after getting into the target lane in preparation for the right turn.

6.3 Experiment with the Blocks World

Although the Blocks World is far less dynamic than in-city driving, it lends itself to scaling studies that involve generalization to tasks with varying numbers of objects. For this domain, we provided ICARUS with the four primitive skills in Table 2 and 11 concepts that were sufficient, in principle,

Table 4. Five skill clauses learned for in-city driving, along with a primitive skill for the same domain.

```
; highest-level skill clause for situations that require a lane change
(driving-in-segment (?me ?g994 ?g1021)  47
 :percepts ((segment ?g994) (lane-line ?g1021) (self ?me))
 :start    ((in-segment ?me ?g994) (steering-wheel-straight ?me))
 :ordered  ((in-lane ?me ?g1021) (centered-in-lane ?me ?g994 ?g1021)
            (aligned-with-lane-in-segment ?me ?g994 ?g1021)
            (steering-wheel-straight ?me)))
; highest-level skill clause for situations that require right turns
(driving-in-segment (?me ?g998 ?g1008)  46
 :percepts ((segment ?g998) (lane-line ?g1008) (self ?me))
 :start    ((steering-wheel-straight ?me))
 :ordered  ((in-segment ?me ?g998) (centered-in-lane ?me ?g998 ?g1008)
            (aligned-with-lane-in-segment ?me ?g998 ?g1008)
            (steering-wheel-straight ?me)))
; skill clause for handling right turns
(in-segment (?me ?g998)  44
 :percepts ((self ?me) (intersection ?g978) (segment ?g998))
 :start    ((last-lane ?g1021))
 :ordered  ((in-intersection-for-right-turn ?me ?g978)
            (steer-for-right-turn ?me ?g978 ?g998)))
; skill clause that prepares the agent for a right turn
(in-rightmost-lane (?me ?g1021)  45
 :percepts ((self ?me) (lane-line ?g1021))
 :start    ((last-lane ?g1021))
 :ordered  ((driving-in-segment ?me ?g994 ?g1021)))
; recursive skill clause that takes first step needed for right turn
(in-intersection-for-right-turn (?me ?g978)  48
 :percepts ((lane-line ?g1021) (self ?me) (intersection ?g978))
 :start    ((last-lane ?g1021))
 :ordered  ((in-rightmost-lane ?me ?g1021)
            (in-intersection-for-right-turn ?me ?g978)))
; primitive skill that is called recursively by learned clause
(in-intersection-for-right-turn (?self ?int)  49
 :percepts ((self ?self) (segment ?sg) (intersection ?int)
            (lane-line ?lane segment ?sg))
 :start    ((in-rightmost-lane ?self ?lane))
 :requires ((in-segment ?self ?sg) (intersection-ahead ?int)
            (last-lane ?lane))
 :actions  ((*cruise))
 :effects  ((in-intersection-for-right-turn ?self ?int)))
```
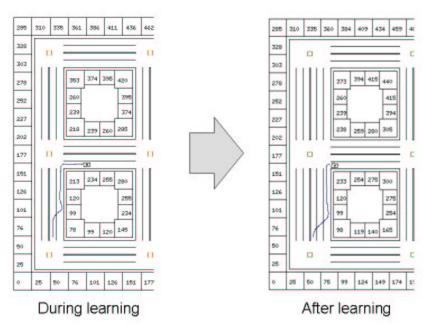
Figure 3. A trace of the ICARUS driving agent's behavior, during and after learning, on a task that required changing to the rightmost lane and turning at the intersection. The trace demonstrates generalization to a new setting with a different number of lanes.

to solve any problem. We then presented the agent with the problems in sequence, using each task as a training problem but also recording the number of cycles and CPU time required to complete it. Because misguided search combined with execution can lead the problem solver into undesirable physical states, we told it to halt if it had not finished a run within 100 cycles and to start over from the initial state. However, the agent could attempt a given problem only five times, and thus spend at most 500 cycles before giving up entirely. We also limited the stack depth to ten goal elements. We enforced these constraints for reasons of practicality and because we think they reflect the manner in which humans tackle novel problems.

We generated randomly a set of random Blocks World tasks that involved settings with 5, 10, 15, 20, 25, and 30 blocks. Each complexity class had 67 to 69 distinct problems, which we ordered by difficulty class (five-block tasks first and 30-block tasks last). The intuition was that the system would learn more effectively if we presented it first with simpler problems, which it could then use in solving more difficult ones. To this end, ICARUS retained skills acquired on successful runs for use in later tasks. We provided the system some 400 randomly generated problem orders and recorded the number of cycles and CPU times needed for each task. As a control, we also ran the system with its learning mechanisms off for another 400 problem sets that were ordered randomly within difficulty classes. Because the problems require different amounts of effort, traditional learning curves are not very informative. Instead, following Minton (1990), we report *cumulative* cycles and CPU times as a function of the number of training problems.

Figure 4 shows the resulting curves, including 95 percent confidence intervals around each mean. As expected, the curves mainly take the form of superlinear functions whose slopes increase with problem difficulty. Although the large scale of plots made the learning and non-learning curves look
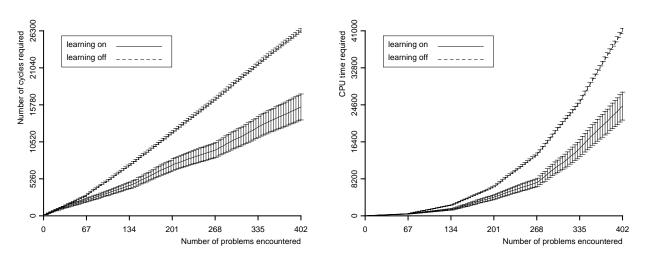
Figure 4. Cumulative number of cycles and CPU times required by ICARUS to solve a Blocks World task as a function of the number of problems encountered, averaged over 400 runs and with problems ordered by difficulty, with the goal stack of size ten. Tick marks on the horizontal axis indicate shifts in problem complexity.

similar for early parts of the curves, there was some benefit for learning even from the beginning, but the difference grows substantially as the systems encounter harder problems. Clearly, prior experience reduces search substantially when it reaches problems with many blocks, and there is no evidence that learning produces a utility problem. Remember that we have made the transfer of learned knowledge challenging in that none of the problems are isomorphic, although they may involve isomorphic subtasks. The results indicate that ICARUS can take advantage of this similar substructure to reduce its effort on later problems.

6.4 Experiment with FreeCell Solitaire

To ensure that our conclusions held for more than the Blocks World, we carried out a similar experiment with FreeCell solitaire, which we described earlier in this section. We gave the ICARUS agent only the 12 basic skills needed to move cards and the top-level goal of getting all cards into foundation cells, along with 24 concepts for describing situations. Unlike the Blocks World, this domain has only one goal condition, but it still has many possible starting states.

For this study, we randomly generated 20 problems each that involved 8, 12, 16, 20, and 24 cards.[5] We ran the system on 300 different sequences of tasks, with simpler problems being presented earlier but ordered randomly within each of the five difficulty classes. As before, we expected that the agent would learn skills from the easier problems that would assist on the harder ones, thus reducing problem-solving effort. For comparison, we presented another 300 random sequences to a non-learning system with the same initial skills and concepts.

Figure 5 presents the cumulative results for this experiment, with error bars that indicate the 95% confidence intervals. As in the Blocks World, the difference between the learning and non-learning conditions is substantial. However, problems with 20 cards or more require a different

---

5. ICARUS' problem solver has difficulty with FreeCell tasks that involve 30 or more cards, apparently because they involve goal interactions that basic means-ends analysis cannot handle.
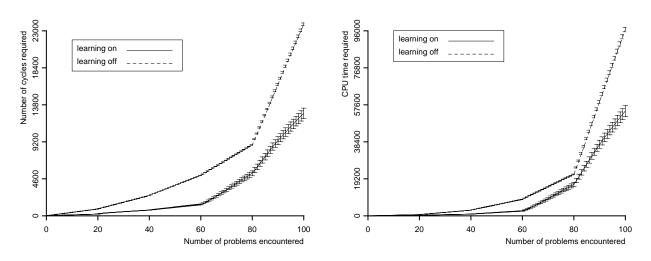
Figure 5. Cumulative number of cycles and CPU times required by ICARUS to solve a FreeCell task as a function of the number of problems encountered, averaged over 300 runs and with problems ordered by difficulty.

class of skills that involve column-to-column moves, which caused the lessened gap between the two conditions around the 80th problem. However, once they have been acquired, these skills provide some advantage, as evidenced by the downturn in the curve for the learning system on the far right of the graphs. Again, we detected no sign of a utility problem as the agent accumulates knowledge in this domain.

7. Related Research

Research on learning cognitive skills from problem solving has a long history within both AI and cognitive science. For example, work on explanation-based learning often aimed to improve efficiency on problem-solving tasks and combined experience with a domain theory to create new cognitive structures. Some techniques focused on the acquisition of search-control rules to guide problem solving, but other efforts dealt instead with the construction of macro-operators from primitive operators (e.g., Iba, 1988; Mooney, 1989; Shavlik, 1989). Our approach to skill learning comes closer to the second paradigm, since both involve composing knowledge elements into larger structures. However, ICARUS adapts this idea to the creation of disjunctive and even recursive skill hierarchies, whereas traditional methods emphasized the creation of 'fixed-sequence' macro-operators that were far less flexible.

ICARUS also bears some similarity to other cognitive architectures that incorporate varieties of analytical or explanation-based learning. For example, Laird, Rosenbloom, and Newell's (1986) SOAR revolves around a problem solver that proceeds until the system encounters an impasse, in which case it creates a subgoal to resolve it. This resolution may require search and take some time to produce the information necessary. Once the impasse has been handled, SOAR creates a *chunk* that encodes a generalized explanation of the result in terms of the original goal context. Intermediate steps from the solution are lost, but the acquired chunk lets the system sidestep similar impasses in the future.

Anderson's (1993) ACT-R employs a related mechanism, called *compilation*, which creates new production rules from ones that are involved in the same reasoning chain. This scheme produces very specific rules that replace variables with the declarative elements against which they matched, rather than forming generalized structures, as do ICARUS and most other systems that learn macro-operators or search-control rules. In fact, our approach is more akin to the composition process that played a role in earlier versions of ACT (Neves & Anderson, 1981), though this mechanism produced fixed behavioral sequences rather than flexible skill hierarchies.

ICARUS' closest architectural relative is PRODIGY (Minton, 1990), which invokes means-ends analysis to solve problems and uses an analytical method to learn either search-control roles or macro-operators from problem-solving traces. Veloso and Carbonell (1993) also describe an extension that records these traces in memory and solves new problems by derivational analogy with earlier ones. None of these mechanisms generates explicit hierarchical structures, but Veloso and Carbonell's approach provides flexibility similar to that found in ICARUS, and the two systems record and utilize very similar information in their goal stacks.

Some other systems support learning in problem-solving domains without making strong architectural commitments. Ruby and Kibler's (1991) SteppingStone learns generalized rules for decomposing complex problems into simpler ones, which it obtains through mixed use of existing problem-reduction rules and forward-chaining exhaustive search when it reaches an impasse. Marsella and Schmidt's (1993) system also acquires task-decomposition rules that incorporate partial orderings among components. Their system combines forward and backward search to identify candidate state pairs, which in turn produce hypothesized problem-reduction rules that are revised based on further experience.[6]

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 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 hierarchical structure. X-Learn also uses a quite sophisticated mixture of analytical and inductive techniques to determine conditions on skills, rather than the much simpler method that ICARUS incorporates.

Another key difference from X-Learn, PRL, and Steppingstone is that ICARUS learns skills for use in reactive execution rather than for use in planning. There has been other work on this topic, but it has emphasized the acquisition of flat controllers rather than hierarchical structures. For instance, Benson's (1995) TRAIL learns teleoreactive controllers for physical agents, but it invokes inductive logic programming to determine rules for individual actions. 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) considers the related task of learning hierarchical controllers, but his formal analysis assumes the agent is provided with annotated sample solutions rather than being generated through problem solving.

---

6. Ilghami et al. (2002) present another system that organizes plan knowledge in a hierarchical manner, but it learns conditions for clause selection rather than the structure of the hierarchy itself.

Other researchers have built systems that support cumulative learning outside the context of problem-solving tasks. One early example was Sammut and Banerji's (1986) Marvin, which learns increasingly complex logical concepts that are composed of ones it has mastered previously. Stone and Veloso (2000) take a similar approach to learning concepts and controllers for playing robotic soccer, although their system acquires quite different types of structure at each level of description. Stracuzzi and Utgoff's (2002) STL algorithm receives training cases about many concepts in parallel, but it learns complex ones only when it has acquired simpler structures that let it master them with little effort. Pfleger (2004) describes another system that acquires hierarchical patterns in an on-line setting, in this case from unsupervised data. Like Marvin and STL, it learns conceptual structures from the bottom up, so that more complex patterns are apparent after simpler ones have been acquired.

8. Concluding Remarks

In the preceding pages, we presented ICARUS, a cognitive architecture for physical agents that uses stored concepts and skills, both organized in hierarchies, to recognize familiar situations and control behavior. We described a new module that supports means-ends problem solving on novel tasks, along with a learning mechanism that produces new skills from traces of problem solutions. This method operates in an incremental manner, creating hierarchical structures that refer to others learned earlier. In addition, we reported experiments with in-city driving, the Blocks World, and FreeCell that showed such learning enables more effective behavior on unfamiliar problems than solving them with only basic knowledge about the domain.

Despite these advances, our work on skill learning in ICARUS is still in its early stages. For instance, we should demonstrate its ability to learn hierarchical structures both on traditional cognitive tasks like multi-column subtraction and on other dynamic domains that, like in-city driving, require the integration of problem solving with reactive control. Future work on driving should show that our methods are sufficient to acquire more complicated skills that involve extended tasks like package delivery and complex settings that include other vehicles. Another promising class of domains for studying skill learning involves two-person games like chess, which seem certain to introduce new challenges because of their extended duration.

In addition, ICARUS' methods for problem solving and hierarchical learning would benefit from new capabilities. We noted earlier that the current system does not chain backward from the start conditions of learned skill clauses. Extending the problem solver to support this ability would mean defining new concepts that characterize the situations in which learned skills are applicable. This addition would also remedy another limitation of the current system, namely its inability to account for the origin of concept hierarchies, which it assumes are given. Such an extension would be straightforward for some tasks, but others will require the ability to acquire recursive concepts. Augmenting the system in this manner may also lead to a utility problem, not during execution of learned skills but during the problem solving used for their acquisition, which we would then need to overcome.

Another drawback is the architecture's reliance on purely deductive inference, which differs markedly from the probabilistic approach taken by its earliest ancestor (Langley et al., 1991). Future versions of the framework should extend the representation of concepts and skills to incorporate probabilities, replace deductive processes with abductive methods that make plausible default inferences, and augment problem solving to operate over skills with uncertain outcomes. We hypothesize that the current mechanisms for learning the structure of skills can be adapted easily to this setting, but we should also introduce methods for estimating the probabilities that annotate the symbolic structures.

We should also note that, although our approach learns skills that generalize to situations with different numbers of objects, its treatment of goals is less flexible. For example, ICARUS can acquire a general procedure for clearing a block that does not depend on the number of blocks above it, but it cannot learn a procedure for constructing a tower with arbitrarily specified components. Extending the method's ability to learn about such recursive goal structures is another important direction for future research that will bring the architecture into closer alignment with the abilities observed in complex human learning.

Acknowledgements

References

Anderson, J. R. (1993). *Rules of the mind*. Hillsdale, NJ: Lawrence Erlbaum.

Asgharbeygi, N., Nejati, N., Langley, P., & Arai, S. (2005). Guiding inference through relational reinforcement learning. *Proceedings of the Fifteenth International Conference on Inductive Logic Programming*. Bonn, Germany: Springer.

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.

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

Doyle, J. (1979). A truth maintenance system. *Artificial Intelligence*, *12*, 231–272.

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.

Forgy, C. L. (1982). Rete: A fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence*, *19*, 17–37.

Jones, R. M., & Langley, P. (in press). A constrained architecture for learning and problem solving. *Computational Intelligence*.

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

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

Khardon, R. (1999). Learning to take actions. *Machine Learning*, *35*, 57–90.

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

Langley, P., Cummings, K., & Shapiro, D. (2004). Hierarchical skills and cognitive architectures. *Proceedings of the Twenty-Sixth Annual Conference of the Cognitive Science Society* (pp. 779–784). Chicago, IL.

Langley, P., McKusick, K. B., Allen, J. A., Iba, W. F., & Thompson, K. (1991). A design for the ICARUS architecture. *SIGART Bulletin*, *2*, 104–109.

Langley, P., & Stromsten, S. (2000). Learning context-free grammars with a simplicity bias. *Proceedings of the Eleventh European Conference on Machine Learning* (pp. 220–228). Barcelona: Springer-Verlag.

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

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.

Nardi, D., & Brachman, R. J. (2002). An introduction to description logics. In F. Baader et al. (Eds.), *Description logic handbook*. Cambridge: Cambridge University Press.

Newell, A. (1990). *Unified theories of cognition*. Cambridge, MA: Harvard University Press.

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.

Pfleger, K. (2004). On-line cumulative learning of hierarchical sparse n-grams. *Proceedings of the Third International Conference on Development and Learning*. San Diego, CA: IEEE Press.

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.

Richman, H. B., Staszewski, J. J., & Simon, H. A. (1995). Simulation of expert memory using EPAM IV. *Psychological Review*, *102*, 305–330.

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., & 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.

Shapiro, D., Langley, P., & Shachter, R. (2001). Using background knowledge to speed reinforcement learning in physical agents. *Proceedings of the Fifth International Conference on Autonomous Agents* (pp. 254–261). Montreal: ACM Press.

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.

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

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

Appendix: Concepts and Skills Provided in Experiments

Table 5. Concepts and skills provided to ICARUS for the in-city driving domain, with italics denoting the goal concept and parentheses indicating the number of clauses for disjunctive skills.

primitive concepts (15)	nonprimitive concepts (26)	primitive skills (19)
stopped	parked	in-intersection-for-right-turn
moving	aligned-with-lane-in-segment	aligned-with-lane-in-segment
in-segment	centered-in-lane	steering-wheel-straight
in-intersection-for-right-turn	steering-wheel-not-straight	centered-in-lane (2)
in-intersection	*driving-in-segment*	in-lane (2)
intersection-ahead	at-speed-for-right-turn	stopped
segment-to-right	ready-for-right-turn	moving
on-right-side-of-road-in-segment	in-leftmost-lane	adjust-speed-for-cruise
in-lane	lane-to-right	adjust-speed-for-right-turn (2)
steering-wheel-straight	lane-to-left	get-on-right-side-of-road
at-speed-for-cruise	in-rightmost-lane	cruise-within-segment
slow-for-right-turn	in-right-turn-lane	steer-for-right-turn
fast-for-right-turn	off-centered-to-right-in-segment	change-lane-to-right
first-lane	off-centered-to-left-in-segment	change-lane-to-left
last-lane	building-on-right	cruise-into-intersection
	building-on-left	cruise
	current-building	
	start-aligned-with-lane-in-segment	
	start-centered-in-lane-1	
	start-centered-in-lane-2	
	start-adjust-speed-for-cruise	
	start-cruise-within-segment	
	start-change-lane-to-right	
	start-change-lane-to-left	
	start-in-lane-1	
	start-in-lane-2	

Table 6. Concepts and skills provided to ICARUS for the Blocks World, with goal concepts in italics.

primitive concepts (4)	nonprimitive concepts (7)	primitive skills (4)
on ontable holding hand-empty	*clear* *three-tower* *two-tower-one-on-table* unstackable pickupable stackable putdownable	unstack pickup stack putdown

Table 7. Concepts and skills provided to ICARUS for FreeCell solitaire, with goal concept in italics.

primitive concepts (10)	nonprimitive concepts (14)	primitive skills (12)
starthome successor colcolpair available-cell available-column clear on bottom incell home	highest *game-won* column-to-home-able column-to-newhome-able column-to-freecell-able lastcolumn-to-home-able lastcolumn-to-newhome-able lastcolumn-to-freecell-able freecell-to-home-able freecell-to-newhome-able freecell-to-column-able column-to-column-able freecell-to-new-column-able column-to-new-column-able	column-to-home column-to-newhome column-to-freecell lastcolumn-to-home lastcolumn-to-newhome lastcolumn-to-freecell freecell-to-home freecell-to-newhome freecell-to-column column-to-column freecell-to-new-column column-to-new-column