72. Simon, H.A., Search and reasoning in problem solving, *Artificial Intelligence* 21 (1983) 7–29.
73. Smith, B.C., Reflection and semantics in a procedural language, MIT/LCS/TR-272, Laboratory for Computer Science, MIT, Cambridge, MA, 1982.
74. Smith, D.E. and Genesereth, M.R., Ordering conjunctive queries, *Artificial Intelligence* 26 (1985) 171–215.
75. van de Brug, A., Rosenbloom, P.S. and Newell, A., Some experiments with R1-Soar, Computer Science Department, Carnegie-Mellon University, Pittsburgh, PA, 1986.
76. van de Brug, A., Bachant, J. and McDermott, J., The taming of R1, *IEEE Expert* 1 (1986) 33–39.
77. VanLehn, K., Felicity conditions for human skill acquisition: Validating an AI-based theory, Xerox Palo Alto Research Center, Palo Alto, CA, 1983.
78. Waterman, D.A. and Hayes-Roth, F. (Eds.), *Pattern Directed Inference Systems* (Academic Press, New York, 1978).
79. Wilensky, R., *Planning and Understanding: A Computational Approach to Human Reasoning* (Addison-Wesley, Reading, MA, 1983).

Received September 1983 with the title "A Universal Weak Method"; revised version received November 1986

CHAPTER 16

Knowledge Level Learning in Soar

P. S. Rosenbloom, Stanford University, J. E. Laird, University of Michigan, and A. Newell, Carnegie Mellon University

Abstract

In this article we demonstrate how knowledge level learning can be performed within the Soar architecture. That is, we demonstrate how Soar can acquire new knowledge that is not deductively implied by its existing knowledge. This demonstration employs Soar's chunking mechanism – a mechanism which acquires new productions from goal-based experience – as its only learning mechanism. Chunking has previously been demonstrated to be a useful symbol level learning mechanism, able to speed up the performance of existing systems, but this is the first demonstration of its ability to perform knowledge level learning. Two simple declarative-memory tasks are employed for this demonstration: recognition and recall.

I. Introduction

Dietterich has recently divided learning systems into two classes: *symbol level learners* and *knowledge level learners* [3]. The distinction is based on whether or not the knowledge in the system, as measured by a knowledge level analysis [10], increases with learning. A system performs symbol level learning if it improves its computational performance but does not increase the amount of knowledge it contains. According to a knowledge level analysis, knowledge only increases if a fact is added that is not implied by the existing knowledge; that is, if the fact is not in the deductive closure of the existing knowledge. Explanation-based generalization (EBG) [2; 9] is a prime example of a learning technique that has proven quite successful as a mechanism for enabling a system to perform symbol level learning. EBG allows tasks that a system can already perform to be reformulated in such a way that they can be performed more efficiently. Because EBG only generates knowledge that is already within the deductive closure of its current knowledge base, it does no knowledge level learning (at least when used in any obvious ways).

Symbol level learning can be quite useful for an intelligent system. By speeding up the system's performance, it allows the system to perform more tasks while using the same amount of resources, and enables the system to complete tasks that capacity limitations previously prevented it from completing. However, an intelligent system cannot live by symbol level learning alone. If a system were incapable of performing knowledge level learning – that is, of adding facts not already implied by its existing knowledge – a host of critical capabilities would be beyond its grasp. These range from relatively simple declarative memory capabilities, such as learning to recognize previously seen objects and to store and retrieve representations of new objects, to more complex capabilities of learning to perform novel tasks.

Soar is an attempt to build an architecture that can support general intelligent behavior [6]. It contains a single learning mechanism, *chunking* [7]. Chunking creates new productions, or chunks, based on the results of goal-based problem solving. The actions of a chunk contain the results of the goal. The conditions of the chunk test those aspects of the pre-goal situation that were relevant to the generation of the results. We have been successful in demonstrating Soar's ability to acquire a variety of types of knowledge, including search-control productions, macro-operators, and operator-implementation productions [14]. We have also demonstrated Soar's ability to perform the basic tasks of EBG and, in the process, provided a mapping between EBG and Soar [12]. This mapping suggests that chunking is a symbol level learning mechanism; an identification that is supported by the fact that all of the demonstrations listed above involve only symbol level learning.

However, chunking originated in psychological theories of the structure of declarative memory [8]. One classical chunking result is that if a list of items is to be memorized for later recall, the memory structure is organized as a hierarchical structure of chunks [1]. Chunking, at least of this classical variety, is thus strongly implicated in the acquisition of new knowledge. This has encouraged us to believe that Soar's chunking mechanism should be able to perform knowledge level learning. It has also placed a responsibility on us to demonstrate these classical chunking phenomena in Soar in order to justify that its learning mechanism is entitled to its name. To distinguish between the form of chunking currently performed by Soar and the more declarative classical chunking capability, we refer to the classical form of chunking as *data chunking*.

The purpose of this article is to report on recent work with Soar that demonstrates data chunking, and thus knowledge level learning, using chunking as the only learning mechanism. In Section II we give a brief overview of Soar. In Section III we describe the fundamental concepts underlying the implementation of data chunking in Soar. In Sections IV and V we then describe in detail how Soar performs two simple data chunking tasks: learning to recognize new objects and learning to recall new objects. In Section VI we conclude

[1]This research was sponsored by the Defense Advanced Research Projects Agency (DOD) under contract N00039-86-C-0133 and by the Sloan Foundation. Computer facilities were partially provided by NIH grant RR-00785 to Sumex-Aim. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency, the US Government, the Sloan Foundation, or the National Institutes of Health.

and describe some important future work.

II. Overview of Soar

Soar is based on formulating all goal-oriented processing as search in problem spaces. The problem space determines the set of legal states and operators that can be used during the processing to attain a goal. The states represent situations. There is an initial state, representing the initial situation, and a set of desired states that represent the goal. An operator, when applied to a state in the problem space, yields another state in the problem space. The goal is achieved when one of the desired states is reached as the result of a string of operator applications starting from the initial state.

Goals, problem spaces, states, and operators exist as data structures in Soar's working memory – a short-term declarative memory. Each goal defines a problem solving context ("context" for short). A context is a data structure in the working memory that contains, in addition to a goal, roles for a problem space, a state, and an operator. Problem solving for a goal is driven by the acts of selecting problem spaces, states, and operators for the appropriate roles in the context. Each of the deliberate acts of the Soar architecture – a selection of a problem space, a state or an operator – is accomplished via a two-phase decision cycle. First, during the elaboration phase, the description of the current situation (that is, the contents of working memory) is elaborated with relevant information from Soar's production memory – a long-term procedural memory. The elaboration process involves the creation of new objects, the addition of knowledge about existing objects, and the addition of preferences. There is a fixed language of preferences that is used to describe the acceptability and desirability of the alternatives being considered for selection. By using different preferences, it is possible to assert that a particular problem space, state, or operator is acceptable (should be considered for selection), rejected (should not be considered for selection), better than another alternative, and so on. When the elaboration phase reaches quiescence – that is, no more productions can fire – the preferences in working memory are interpreted by a fixed decision procedure. If the preferences uniquely specify an object to be selected for a role in a context, then a decision can be made, and the specified object becomes the current value of the role. The decision cycle then repeats, starting with another elaboration phase.

If an elaboration phase ever reaches quiescence while the preferences in working memory are either incomplete or inconsistent, an impasse occurs in problem solving because the system does not know how to proceed. When an impasse occurs, a subgoal with an associated problem solving context is automatically generated for the task of resolving the impasse. The impasses, and thus their subgoals, vary from problems of selection (of problem spaces, states, and operators) to problems of generation (e.g., operator application). Given a subgoal, Soar can bring its full problem solving capability and knowledge to bear on resolving the impasse that caused the subgoal. When subgoals occur within subgoals, a goal hierarchy results (which therefore defines a hierarchy of contexts). The top goal in the hierarchy is a task goal. The subgoals below it are all generated as the result of impasses in problem solving. A subgoal terminates when its impasse is resolved, even if there are many levels of subgoals below it (the lower ones were all in the service of the terminated subgoal, so they can be eliminated if it is resolved).

Chunking is a learning mechanism that automatically acquires new productions that summarize the processing that leads to results of subgoals. The actions of the new productions are based on the results of the subgoal. The conditions are based on those aspects of the pre-goal situation that were relevant to the determination of the results. Relevance is determined by treating the traces of the productions that fired during the subgoal as dependency structures. Starting from the production trace that generated the subgoal's result, those production traces that generated the working-memory elements in the condition of the trace are found, and then the traces that generated their condition elements are found, and so on until elements are reached that exist outside of the subgoal. These elements form the basis for the conditions of the chunk. Productions that only generate preferences do not participate in this backtracing process – preferences only affect the efficiency with which a goal is achieved, and not the correctness of the goal's results. Once the working-memory elements that are to form the basis of the conditions and actions of a chunk have been determined, the elements are processed to yield the final conditions and actions. For the purposes of this article, the most important part of this processing is the replacement of some of the symbols in the working-memory elements by variables. If a symbol is an object identifier – a temporary place-holder symbol used to tie together the information about an object in working memory – then it is replaced by a variable; otherwise the symbol is left as a constant. This is the minimal generalization required to get any transfer.

Chunking applies to all of the subgoals generated during task performance. Once a chunk has been learned, the new production will fire during the elaboration phase in relevantly similar situations in the future, directly producing the required information. No impasse will occur, and problem solving can proceed smoothly. Chunking is thus a form of goal-based caching which avoids redundant future effort by directly producing a result that once required problem solving to determine.

III. Fundamentals of Data Chunking

Reduced to its essentials, data chunking involves the perception of some new piece of knowledge, followed by the storage of some representation of the new knowledge into long-term memory. Thus, the first step in performing data chunking in Soar is for Soar to use its perceptual capabilities to generate a representation of the new knowledge in its working memory.[2] At this point, the new knowledge is available for use by the system, but it has not yet been learned – working memory is only a temporary memory which holds the current data upon which the system is working. The learning act occurs when a production is created which can, at appropriate points in the future, retrieve the new knowledge into working memory. If Soar is to use its chunking mechanism to do this, it must take advantage of the fact that chunking learns from goal-based experience. The key is for it to set up the right internal tasks so that its problem solving experience in subgoals leads to the creation of chunks that represent the new knowledge. Suppose Soar is to memorize a new object, call it object A, so that it can be recalled on demand. To accomplish this, a chunk needs to be acquired that can generate the object when the demand arises. The appropriate internal task for this problem would appear to be simply to copy the object in a subgoal. The chunk that is learned from this experience has actions which generate an object B that is a copy of object A.

[2] Soar does not yet have an appropriate I/O interface, so in the current implementation of data chunking this perceptual phase is performed by special purpose Lisp code.

This simple solution glosses over two important problems. The first problem is that, if the generation of object B is based on an examination of object A, then the conditions of the chunk will test for the existence of object A before generating object B, thus allowing the object to be recalled in only those circumstances where it is already available. The solution to this problem that we have discovered is to split the act of recalling information into separate generate and test phases. A generation problem space is provided in which new objects can be constructed by generating and combining objects that the system has already learned to recall. Object B is thus constructed from scratch out of objects that the system already knows, rather than being a direct copy of object A. Object A is not examined during this process; instead, it is examined during a test phase in which it is compared with object B to see if they are equivalent. Separate chunks are learned for the generate and test phases, allowing a chunk to be learned that generates object B without examining object A.

The second problem is that, at recall time, the system must both generate the learned object B and avoid generating all of the other objects that it could potentially generate. The direct effect of the generation chunk is simply to cache the generation of object B, allowing it to be generated more efficiently in the future (symbol level learning). This, by itself, does not enable Soar to discriminate between object B and the other objects that could be generated (knowledge level learning). However, this additional capability can be provided if: all of the learned objects can be recalled before any new objects can be generated; and if a termination signal can be given after the learned objects have been recalled and before any other objects are generated. In Soar, this capability is provided directly by the structure of the decision cycle. The chunks fire during the elaboration phase, allowing learned objects to be recalled directly. After all of the learned objects have been recalled, an impasse occurs. Other objects could be generated in the subgoal for this impasse, or alternatively (and correctly) the impasse can be treated as a termination signal, keeping other objects from being generated. Soar can thus break through the otherwise seamless interface, in which a cached value looks exactly like a computed value, by making use of Soar's ability to reflect on its own behavior [13] – specifically, its ability to base a decision on whether an impasse has occurred.

Generation chunks thus support symbol level learning (caching the generation of the object) and knowledge level learning (correct performance on recall tasks). As described in the following two sections, rather than actually learning test chunks, recognition chunks are learned. These recognition chunks speed up the performance of the system on both recall and recognition tasks (symbol level learning), plus they allow Soar to perform correctly on recognition tasks (knowledge level learning). The abilities to learn to recognize and recall new objects are two of the most basic, yet most important, data chunking capabilities. If Soar is able to accomplish these two paradigmatic learning tasks, it would seem to have opened the gates to the demonstration of the remaining data chunking tasks, as well as to more sophisticated forms of knowledge level learning.

IV. Recognition

The recognition task is the simplest declarative memory task. There are two types of trials: training and performance. On each training trial the system is presented with a new object, and it must learn enough to be able to perform correctly on the performance trials. On each performance trial the system is presented with an object which it may or may not have seen during the training trials. It must respond affirmatively if it has seen the object, and negatively if it has not.

The objects that the system deals with are one of two types: primitive or composite. Primitive objects are those that the system is initially set up to recognize: the letters a–z, plus the special objects [and]. A composite object is a hierarchical structure of simpler objects that is eventually grounded in primitive objects. The object representation includes two attributes: name and substructure. An object is recognized if it has a name. A primitive object has nothing but a name. A composite object may or may not have a name, depending on whether it is recognized or not. A composite object is distinguished from a primitive object by having a substructure attribute that gives the list of objects out of which the object is composed. The list always begins with [, ends with], and has one or more other objects – either primitive or composite – in between. For example, [a b c] and [[a b c] [d e]], are two typical composite objects.

To learn to recognize a new composite object, an internal task is set up in which the system first recognizes each of the subobjects out of which the object is composed, and then generates a new name for the composite object. The name becomes the result of the subgoal, and thus forms the basis for the action of a chunk. The name is dependent on the recognition of all of the object's subobjects, so the conditions of the chunk test for the subobjects' names. During a performance trial, the recognition chunk can be used to assign a name to a presented object if it is equivalent to the learned one, allowing an affirmative response to be made to the recognition query.

In more detail, a training trial begins with a goal to learn to recognize an object. A recognition problem space is selected along with a state that points to the object that is to be learned – the *current object* – for example, [a b c]. If the current object is recognized – that is, has a name – the training trial is terminated because its task is already accomplished. There is only one operator in the recognition problem space: *get-next-element*. If the current-object is recognized, then the get-next-element operator receives an acceptable preference, allowing it to be selected as the current operator. When the operator is executed, it generates a new state that points to the object that follows the current one.

However, if the current object is not recognized, the get-next-element operator cannot be selected, and an impasse occurs. It is in the subgoal that is generated for this impasse that recognition of the object is learned. The recognition problem space is used recursively in this subgoal, with an initial state that points to the object's first subobject (i.e., [). Because this new current object has a name, the get-next-element operator is selected and applied, making the next subobject (a, for the current example) the current object. If the subobject were not recognized, a second-level subgoal would be generated, and the problem solving would again recur, but this time on the substructure of the subobject. The recursion is grounded whenever objects are reached that the system has previously learned to recognize. Initially this is just for the primitive objects, but as the system learns to recognize composite objects, they too can terminate the recursion.

When the system has succeeded in recognizing all of the object's subobjects, a unique internal name, such as *p0045*, is generated for the object. The new name is returned as the result of the subgoal, allowing the problem solving to proceed in the parent context because now its current object has a name. The subgoal is thus terminated, and a chunk is learned that examines the object's subobjects, and generates the object's name. This recognition production can fire whenever a state is selected that points to an object that has the same substructure. In schematic pseudo-code, the production for the current example looks like the following.

```
Current-Object(s, [a b c]<z>) -->
   Name(z, *p0045*)                                    (1)
```

The variable s binds to the current state in the context. The variable x binds to the identifier of the current object, whose substructure must be `[a b c]`. The appearance of the relevant constants – `[`, `a`, `b`, `c`, `]`, and `*p0045*` – in the conditions and actions of this production occur because, in creating a chunk from a set of production traces, constant symbols are not replaced by variables.

If `[a b c]` is now presented on a performance trial, production 1 (above) fires and augments the object with its name. The system can then respond that it has recognized the object because there is a name associated with it. If an unknown object, such as `[x y z]`, is presented on a performance trial, no recognition production fires, and an impasse occurs. This impasse is used as a signal to terminate the performance trial with a "no" answer.

If the object being learned is a multi-level composite object, then in addition to learning to recognize the object itself, recognition productions are learned for all of the unrecognized subobjects (and subsubobjects, etc.). For example, if the system is learning to recognize the object `[[a b c] [d e]]`, it first uses production 1 to recognize `[a b c]` and then learns the following two new recognition productions:

```
Current-Object(s, [d e]<z>) -->
   Name(z, *p0046*)                                    (2)
Current-Object(s, [*p0045* *p0046*]<z>) -->
   Name(z, *p0047*)                                    (3)
```

Chunks are also learned that allow composite subobjects to be recognized directly in the context of the current object. To recognize a composite subobject without these chunks, the system would have to go into a subgoal in which the subobject could itself be made the current object.

If `[[a b c] [d e]]` is now presented on a performance trial, productions first fire to recognize `[a b c]` and `[d e]` as objects `*p0045*` and `*p0046*`. Production 3 then fires to recognize `[*p0045* *p0046*]` as object `*p0047*`. The system can then reply in the affirmative to the recognition query.

V. Recall

The recall task involves the memorization of a set of objects, which are later to be generated on demand. From the point of view of the internal task, it is the dual of the recognition task. Instead of incorporating information about a new object into the conditions of a production, the information must be incorporated into the actions. As with recognition, there are training and performance trials. On each training trial the system is presented with a new object, and it must learn to generate the object on demand. On a performance trial, the system receives a recall request, and must respond by producing the objects that it learned to generate on the training trials.

As described in Section III, on a training trial the general approach is to set up a two-phase internal task in which the object is copied. In the first phase, a new composite object is generated by executing a sequence of operators that recall and assemble subobjects that the system already knows. This generation process does not depend on the presented object. In the second phase, the generated object is tested to see if it is equivalent to the presented object. Though this approach solves the problem discussed in Section III, it also introduces a smaller but still important technical issue – how to efficiently generate the new object without examining the presented object. Because it is possible to generate any object that can be constructed out of the already known objects, there is a control problem involved in ensuring that the right object is generated. The solution to this problem is to use the presented object as search-control knowledge during the process of generating the new object. Search-control knowledge determines how quickly a problem is solved, not the correctness of the solution – the goal test determines the correctness – so the result does not depend on any of the knowledge used to control the search. Thus, chunks never incorporate control knowledge. In consequence, the generation process can proceed efficiently, but the chunk created for it will not depend on the presented object.

In more detail, a training trial begins with a goal to learn to recall a presented object. The system selects a recall problem space. An initial state is created and selected that points to the presented object; for example, `Presented(s1, [a b c])`, where `s1` is the identifier of the state. There is only one type of operator in the recall problem space: *recall*. An instance of the recall operator is generated for each of the objects that the system knows how to recall. To enable the system to find these objects, they are all attached to the recall problem space. This can be a very large set if many objects have been memorized; a problem to which we return in Section VI. Initially the system knows how to recall the same primitive objects that it can recognize: a–z, `[`, and `]`. This set increases as the system learns to recall composite objects.

The presented object acts as search control for the generation process by influencing which recall operator is selected. First the system tries to recognize the presented object. For the current example, production 1 fires, augmenting the object with its name (`*p0045*`). If the system had not previously learned to recognize the presented object, it does so now before proceeding to learn to recall it. Then, if there is a recall operator that will recall an object with the same name, an acceptable preference is generated for the operator, allowing it to be selected. When a recall operator executes, it creates a new state in which it adds the recalled object to a structure representing the object being generated. If this happens in the top goal, it means that the system has already learned to recall the presented object, and it is therefore done with the training trial.

However, when the system does not already know how to recall the object, as is true in this instance, no recall operator can be selected. An impasse occurs and a subgoal is generated. In this subgoal, processing recurses with the attempt to recall the subobjects out of which the presented object is composed. A new instance of the recall problem space is created and selected. Then, an initial state is selected that points to the first subobject of the presented object (`Presented(s2, [)`). In this subgoal, processing proceeds just as in the parent goal. If the object is not recognized, the system learns to recognize it. Then, if the object cannot be recalled, the system learns to recall it in a further subgoal. However, in this case the object (`[`) is a primitive and can thus already be recognized and recalled. The appropriate recall operator is selected and creates a new state with a newly generated `[` object in it (`Generated(s3, [)`). The operator also augments the new state with the successor to the presented object (`Presented(s3, a)`). This information is used later to guide the selection of the next recall operator.

The system continues in this fashion until a state is created that contains a completely generated object (for example, `Generated(s7, [a b c])`). The one thing missing from the generated object is a name, so the system next tries to recognize the generated object as an instance of some known object. If recognition fails, the subgoal stays around and the system has the opportunity to try again to generate a recognizable object. If recognition succeeds, as it does here, the generated object is augmented with its name (`*p0045*`). Generation is now complete, so the generated object is added to the set of objects that can be recalled in the parent goal (unless there is already an object with that name in the set). This act makes the generated object a result of the subgoal, causing a chunk to be learned which can generate the object in the future. Execution of this chunk is the basic act of retrieving the remembered object from long-term (production) memory into working memory. In schematic pseudo-code, this chunk looks like the following.

```
-Object(recall, *p0045*) -->
   Object(recall, *p0045*[a b c])                      (4)
```

This production says that the object should be generated and attached to the recall problem space if there is not already an object with that name so attached.

Though generation is now complete, the generated object cannot yet be recalled in the parent goal until a goal test has been performed to ensure that the generated object is equivalent to the presented object. This test is performed by comparing the name of the presented object with the name of the generated object. If the names match, a recall operator can be selected in the parent goal for the generated object, and the subgoal is terminated. The recall operator is then executed, and processing continues. If the names do not match, no recall operator is selected, the subgoal does not terminate, and the system has the opportunity to keep trying.

During a performance trial, the top goal is to recall all of the objects so far learned. A recall problem space is created, selected, and then augmented with the set of objects that the system has learned to recall. Since the goal is to recall all learned objects rather than just a specific one, acceptable and indifferent preferences are created for all of the recall operators, allowing everything that has been so far learned to be recalled in random order – the indifferent preferences state that it doesn't care which of the operators is selected first. Recall performance is terminated when no more recall operators can be selected. This condition is signaled by the occurrence of an impasse. In the resulting subgoal the system could generate more objects, but it should not because they would not correspond to objects it has seen.

If the object being learned is a multi-level composite object, the system learns to recall the object as well as each subobject, assuming it has not previously learned them. If the system were to learn to recall the object `[[a b c] [d e]]`, given that it has already learned to recognize the object and its subobjects, and to recall the subobject `[a b c]`, the following two new generation productions would be learned.

```
-Object(recall, *p0046*) -->
   Object(recall, *p0046*[d e])                        (5)
-Object(recall, *p0047*) -->
   Object(recall, *p0047*[*p0045* *p0046*])            (6)
```

On a performance trial that follows these training trials, the system would recall all three objects.

VI. Conclusion

In this article we have demonstrated how Soar can expand its knowledge level to incorporate information about new objects, and thus perform knowledge level learning. This was accomplished with chunking, a symbol level learning mechanism, as the only learning mechanism. One new mechanism was added to Soar for this work: the ability to generate new long-term symbols to serve as the names of objects. However, this capability is only critical for the learning of object hierarchies. Knowledge level learning can be demonstrated for simpler one-level objects without this added capability.

One implication of this demonstration is that caution must be exercised in classifying learning mechanisms as either symbol level or knowledge level. The distinction may not be as fundamental as it seems. In fact, other symbol level learning mechanisms, such as EBG, may also be able to produce knowledge level learning. A second implication of this demonstration is that chunking may not have been misnamed, and that it may be able to produce the full span of data chunking phenomena.

Three important items are left for future work. The first item is to extend the demonstrations provided here to more complex tasks. Work is currently underway on several projects that incorporate data chunking as part of a larger whole. In one such project, data chunking will be used during the acquisition of problem spaces for new tasks [14]. Work is also underway on more complex forms of knowledge level learning. In one such project, based on the work described in [5], analogical problem solving will be used as a basis for bottom-up (generalization-based) induction. In a second such project, top-down (discrimination-based) induction is performed during paired-associate learning (see also the next paragraph). Both of these latter two projects demonstrate what Dietterich termed *nondeductive knowledge level learning* [3].

The second item is to overcome a flaw in the way recall works. The problem is that whenever a recall problem space is entered, all of the objects that the system has ever learned to recall are retrieved from production memory into working memory. If the system has remembered many objects, this may be quite a time-consuming operation. We have begun work on an alternative approach to recall that is based on a cued-recall paradigm. In this version, the system builds up a discrimination network of cues that tell it which objects should be retrieved into working memory. Early results with this version have demonstrated the ability to greatly reduce the number of objects retrieved into working memory. The results also demonstrate a form of discrimination-based induction that allows objects to be recalled based on partial specifications.

The third item is to use our data chunking approach as the basis for a psychological model of declarative learning and memory. There are already a number of promising indications: the resemblance between our model of recall and the generate-recognize theory of recall (see, for example, [15]); the resemblance between the discrimination network learned during cued recall and the EPAM model of paired-associate

learning [4]; the resemblance of retrieval-by-partial-specification to the description-based memory model of Norman and Bobrow [11]; and the way in which both learning and retrieval are reconstructive processes in the cued recall model. These resemblances came about not because we were trying to model the human data, but because the constraints on the architecture forced us to approach the problems in the way we have.

References

1. Buschke, H. "Learning is organized by chunking." *Journal of Verbal Learning and Verbal Behavior 15* (1976), 313-324.
2. DeJong, G., & Mooney, R. "Explanation-based learning: An alternative view." *Machine Learning 1* (1986), 145-176.
3. Dietterich, T. G. "Learning at the knowledge level." *Machine Learning 1* (1986), 287-315.
4. Feigenbaum, E. A., & Simon, H. A. "EPAM-like models of recognition and learning." *Cognitive Science 8* (1984), 305-336.
5. Golding, A., Rosenbloom, P. S., & Laird, J. E. Learning general search control from outside guidance. Proceedings of IJCAI-87, Milan, 1987. In press
6. Laird, J. E., Newell, A., & Rosenbloom, P. S. "Soar: An architecture for general intelligence." *Artificial Intelligence 33* (1987). In Press
7. Laird, J. E., Rosenbloom, P. S., & Newell, A. "Chunking in Soar: The anatomy of a general learning mechanism." *Machine Learning 1* (1986), 11-46.
8. Miller, G. A. "The magic number seven plus or minus two: Some limits on our capacity for processing information." *Psychological Review 63* (1956), 81-97.
9. Mitchell, T. M., Keller, R. M., & Kedar-Cabelli, S. T. "Explanation-based generalization: A unifying view." *Machine Learning 1* (1986), 47-80.
10. Newell, A. "The knowledge level." *AI Magazine 2* (1981), 1-20.
11. Norman, D. A., & Bobrow, D. G. "Descriptions: An intermediate stage in memory retrieval." *Cognitive Psychology 11* (1979), 107-123.
12. Rosenbloom, P. S., & Laird, J. E. Mapping explanation-based generalization onto Soar. Proceedings of AAAI-86, Philadelphia, 1986.
13. Rosenbloom, P. S., Laird, J. E., & Newell, A. Meta-levels in Soar. Proceedings of the Workshop on Meta-Level Architecture and Reflection, Sardinia, 1986.
14. Steier, D. M., Laird, J. E., Newell, A., Rosenbloom, P. S., Flynn, R., Golding, A., Polk, T. A., Shivers, O. G., Unruh, A., & Yost, G. R. Varieties of Learning in Soar: 1987. Proceedings of the Fourth International Machine Learning Workshop, Irvine, 1987. In press
15. Watkins, M. J., & Gardiner, J. M. "An appreciation of generate-recognize theory of recall." *Journal of Verbal Learning and Verbal Behavior 18* (1979), 687-704.

CYPRESS-Soar: A Case Study in Search and Learning in Algorithm Design

D. M. Steier, Carnegie Mellon University

Abstract

This paper describes a partial reimplementation of Doug Smith's CYPRESS algorithm design system within the Soar problem-solving architecture. The system, CYPRESS-SOAR, reproduces most of CYPRESS' behavior in the synthesis of three divide-and-conquer sorting algorithms from formal specifications. CYPRESS-Soar is based on heuristic search of problem spaces, and uses search to compensate for missing knowledge in some instances. CYPRESS-Soar also learns as it designs algorithms, exhibiting significant transfer of learned knowledge, both within a single design run, and across designs of several different algorithms. These results were produced by reimplementing just the high-level synthesis control of CYPRESS, simulating the results of calls to CYPRESS' deduction engine. Thus after only two months of effort, we had a surprisingly effective research vehicle for investigating the roles of search, knowledge, and learning in this domain.*

I Introduction

Good human programmers have at least two remarkable abilities: they manage to produce programs in the face of incomplete knowledge, and they make use of previous experience in solving new problems. How could we get automatic programming systems to produce the same intelligent behavior? AI-based performance systems in other domains compensate for incomplete knowledge by searching through a space of possible solutions, and there exist a variety of mechanisms for learning from experience. However, automatic programming research has so far produced only a few systems that either search or learn, and, to my knowledge, none that do both. This is true despite the field's growing acknowledgement of the importance of both search and learning [2, 3, 4].

This paper describes a prototype system that both searches and learns while performing part of an automatic programming task. An algorithm design system, previously built within a special-purpose framework, was reimplemented in a more general problem-solving architecture with built-in search and learning capabilities. The previously implemented system is Doug Smith's CYPRESS [11, 12, 13], which is most noted for its design of divide-and-conquer algorithms. The foundation for the reimplementation is Soar [7, 8], an architecture for general intelligence developed by John Laird, Allen Newell, and Paul Rosenbloom. The combined system, CYPRESS-Soar, produces the bulk of three of CYPRESS' sorting algorithm derivations, and takes advantage of the properties of Soar to search and learn while doing so.

In Section II, I describe CYPRESS and its approach to the synthesis of divide-and-conquer sorting algorithms and in Section III I give an overview of the Soar architecture. The remaining sections discuss CYPRESS-Soar, presenting the following results:

- **Performance without fixed design strategies:** CYPRESS-Soar uses any knowledge available at run-time to decide when algorithm refinement operators should be applied. If the knowledge is unavailable, CYPRESS-Soar automatically falls back on general problem-solving methods, initiating lookahead search to evaluate the possibilities. In contrast, design strategies control operator application in CYPRESS, and any necessary search must be guided by an expert user. (Section IV)
- **Transfer of learned knowledge:** CYPRESS-Soar knows what goal it is working on, and caches the result of the goal for future use. Because some goals show up more than once, this learning mechanism reduces problem-solving effort, both within the design of a single algorithm, and on later designs of different algorithms. CYPRESS does not learn, and consequently can not take advantage of repeated subgoals. (Section V)

Section VI concludes with a discussion of several issues involved in extending the prototype CYPRESS-Soar system into a more general automatic algorithm designer.

II How CYPRESS designs divide-and-conquer algorithms

CYPRESS is a semi-automatic system that derives algorithms from formal specifications. It works by top-down refinement of program schemes, or templates, which represent abstractions such as *divide-and-conquer* and *generate-and-test*. A problem specification is matched against a program scheme, and with the aid of a design strategy, decomposed into specifications of simpler problems. This problem reduction process continues recursively until a specification can be solved directly by primitive operators known to the system. When more than one design strategy is applicable, or more than one operator matches a specification, the user makes a selection among alternatives.

CYPRESS spends most of its time in calls to RAINBOW, its deduction engine. RAINBOW performs a generalized version of theorem-proving known as *antecedent derivation* [11]. Given a set of hypotheses, H and a goal formula, G, RAINBOW tries to give the weakest possible precondition, or antecedent, P such that the hypotheses in H conjoined with P imply G. If P is just *true*, then G is already a valid formula given H. In the context of algorithm synthesis, RAINBOW is used to reason backwards from output conditions to test if a specification is satisfied. If it is not satisfied, the derived antecedent is used as dictated by the active design strategy as the basis for further action. Viewed in problem-solving terms, RAINBOW provides a sophisticated form of means-ends analysis.

The input to CYPRESS is a formal specification of the problem to be solved, giving the input and output domains (types, or sets), and input and output conditions for the problem. A specification of the problem of sorting lists of natural numbers from [13] is

$$\text{SORT}{:}x = z \text{ such that } \text{Bag}{:}x = \text{Bag}{:}z \wedge \text{Ordered}{:}z$$
$$\text{where SORT: LIST(N)} \rightarrow \text{LIST(N)}.$$

The SORT function maps the input x into the output z. An implicit input condition, *true* is assumed. The output condition is that the bag (multiset) of elements in x is the same as in z, and z is ordered. The specification assumes pre-existing knowledge of the terms "Bag" and "Ordered".

The sorting problem is amenable to a divide-and-conquer solution. The CYPRESS scheme for divide-and-conquer is expressed in a typed functional programming language, a derivative of Backus' FP [1]:

$$\begin{aligned} F{:}x \equiv\ & \textbf{if} \\ & \quad \text{Primitive}{:}x \rightarrow \text{Directly\_Solve}{:}x \ \| \\ & \quad \neg\text{Primitive}{:}x \rightarrow \text{Compose} \cdot (G \times F) \cdot \text{Decompose}{:}x \\ & \textbf{fi} \end{aligned}$$

The scheme abstractly specifies how to compute the value of F on input x: If x is a base-case input, then solve it directly; otherwise, decompose x into two subproblems, recursively solve one and apply an auxiliary function G to the other, then compose the results.

*This research was supported in part by the National Science Foundation under Grant DCR-8412139, and in part by the Defense Advanced Research Projects Agency under Contract F336 15-81-K-1539. Work on Cypress-Soar was begun while the author was visiting Kestrel Institute.