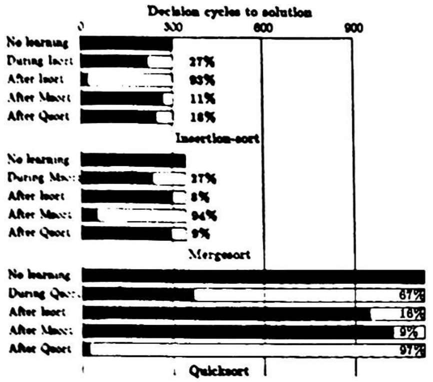

Figure 2: Effects of learning with minimal search control

leads to a large number of similar situations and operator applications, and with the generalization performed by chunking, much redundant problem-solving effort can be saved. This is especially true in in the quicksort synthesis where the system needs to search for the right input condition for partition: the reduction in decision cycles from learning is close to 70%.

In CYPRESS-Soar, the majority of the within-trial transfer results from the need to actually apply an operator after it has been evaluated by lookahead. Since evaluating an operator by lookahead implies computing the result of the operator in the process, after lookahead there will be a chunk that directly creates the new state once the operator is actually selected. The context in which the chunk fires is identical to the one in which the chunk was formed, so the transfer is not very surprising.

The remaining within-trial transfer occurs when an algorithm is synthesized for a subproblem specification in one context, and the same specification shows up again later in another context in the same design. An example of this shows up in the quicksort derivation. In synthesizing partition, CYPRESS-Soar proposed three possible input conditions in searching for the correct one, each time retaining the same output condition. Since the specification for the composition subalgorithm was unaffected by changes in the input condition, the same composition algorithm could be used on each attempt. With minimal search control, the savings from eliminating redundant syntheses of the composition amounted to about one-fifth of the total problem-solving effort of the run without learning. In more complicated algorithms and specifications, one would expect the savings from learning to be even greater.

The last three bars show that CYPRESS-Soar also exhibits *across-trial transfer*, improving performance on subsequent designs of the same algorithm, and across-task transfer, applying knowledge learned from the design of one algorithm to subsequent designs of different algorithms. For example, with full search control and no learning, it took CYPRESS-Soar [illegible] decision cycles to synthesize insertion-sort. As one might expect, it took almost no effort to synthesize quicksort after learning on it, only 20 decision cycles, a savings of [illegible]%. But some of the transfer also occurred after designing the other algorithms: [illegible] decision cycles after mergesort and [illegible] after quicksort, savings of [illegible]% and 18% respectively. Reductions of [illegible]% in solution lengths were observed across all pairs of algorithms, in both the minimal and the full search control runs.

The transfer occurs mostly because all three algorithms solve the same problem, namely sorting. Each sorting algorithm must decompose lists, and so the same well-founded ordering by list length can be preserved by all three top-level decomposition operators. Other transfer occurs in implementing simple induction operators, such as negating certain logical expressions. Also, [illegible] led to transfer between mergesort and quicksort, because in both cases the input is either a single-element or null list. There is no transfer to insertion-sort, because there the input condition specifies only sorting null lists. The representation of the input condition would have to be changed for the matcher, which only fires chunks in the case of an exact syntactic match to the context, to detect that an operator that handles a certain type of input could also handle subsets of that input.

VI Discussion

While a system that designs three algorithms is better than a system that only designs one, CYPRESS-Soar is still not a general automatic algorithm designer, not even within the class of divide-and-conquer algorithms. This is mainly due to the special-case rules for deduction and conditional synthesis, a consequence of the strategic choice in this research to focus first on search and learning in a few divide-and-conquer algorithms. For a general system, one would need to implement additional problem spaces that would perform these functions. We foresee no theoretical barriers to such extensions. The major hurdles to be dealt with are the construction of better interfaces for working with logical formulae in Soar, and the efficiency of a Soar-based deduction engine.

Perhaps most important is that with the existing chunks and the ability to precisely measure across-task transfer, CYPRESS-Soar forms a unique experimental vehicle with which to explore the potential for learning in this domain. The degree to which a Soar-based system can apply chunks to improve its performance depends on how often similar situations are repeated as subgoals while problem-solving. The repetition may be less frequent than it could be because CYPRESS-Soar does not currently break down the deduction into subgoals. It is likely that more transfer would occur if the deduction engine were implemented completely within Soar. More fundamentally the representation used by CYPRESS-Soar may need to capture abstractions common to the algorithms in the syntax of the representation language. On the other hand, it may be the case with these sorting algorithms that no further transfer is possible; that the design processes needed for their creation are just not very similar.

While much work remains to be done, it is encouraging that the current results were obtained in CYPRESS-Soar with only two months' work. This demonstration that a formal theory of design is fully compatible with a general framework for intelligent action was possible only because of the strong foundations available in the work on CYPRESS and Soar. It is also encouraging that the issues raised in the course of developing CYPRESS-Soar have seemed to be worthwhile research topics; in addressing them, we expect to gain useful insights about algorithms and the processes involved in their design.

Acknowledgements

I am most grateful to Doug Smith and the Kestrel Institute for providing me with the opportunity and the environment to begin this research, and to Allen Newell for numerous discussions during the development of CYPRESS-Soar. Doug Smith, Allen Newell, Elaine Kant, John Laird, Craig Knoblock, Dorothy Setliff and Oren Etzioni also made useful comments on earlier drafts of this paper.

References

1. Backus, J. "Can programming be liberated from the von Neumann style?: a functional style of programming and its algebra of programs". *Communications of the ACM 21*, 8 (August 1978), 613-641.
2. Balzer, R. "A 15-year perspective on automatic programming". *IEEE Transactions on Software Engineering SE-11*, 11 (November 1985).
3. Barstow, D. R. "Domain-specific automatic programming". *IEEE Transactions on Software Engineering SE-11*, 11 (November 1985).
4. Dietzen, S. R. and Scherlis, W. L. Analogy in program development. Proceedings of the Second Conference on the Role of Language in Problem Solving, April, 1986.
5. Laird, J. E. Universal subgoaling. In *Universal Subgoaling and Chunking: The Automatic Generation and Learning of Goal Hierarchies*, Kluwer Academic Publishing, Hingham, MA, 1986.
6. Laird, J. E., Newell, A., and Rosenbloom, P. S. "Soar: An architecture for general intelligence". *Artificial Intelligence* (1987). in press.
7. Laird, J. E., Rosenbloom, P. S., and Newell, A. Towards chunking as a general learning mechanism. Proceedings of AAAI-84, The American Association for Artificial Intelligence, Austin, Texas, August, 1984, pp. 188-192.
8. Laird, J. E., Rosenbloom, P. S., and Newell, A. "Chunking in Soar: The anatomy of a general learning mechanism". *Machine Learning 1*, 1 (1986).
9. Rosenbloom, P. S. The chunking of goal hierarchies. In *Universal subgoaling and chunking: The automatic generation and learning of goal hierarchies*, Kluwer Academic Publishing, Hingham, MA, 1986.
10. Rosenbloom, P. S., Laird, J. E., McDermott, J., Newell, A., and Orciuch, E. "R1-Soar: An experiment in knowledge-intensive programming in a problem-solving architecture". *IEEE Transactions on Pattern Analysis and Machine Intelligence 7*, 5 (1985), 561-569.
11. Smith, D. R. Derived preconditions and their use in program synthesis. In *Sixth Conference on Automated Deduction*, Springer-Verlag, 1982. Lecture Notes in Computer Science 138.
12. Smith, D. R. "The design of divide-and-conquer algorithms". *Science of Computer Programming 5* (1985), 37-58.
13. Smith, D. R. "Top-down synthesis of divide-and-conquer algorithms". *Artificial Intelligence 27*, 1 (1985), 43-96.

Varieties of Learning in Soar: 1987

D. M. Steier, Carnegie Mellon University, J. E. Laird, University of Michigan,
A. Newell, Carnegie Mellon University, P. S. Rosenbloom, Stanford University,
R. Flynn, Carnegie Mellon University, A. Golding, Stanford University,
T. A. Polk, Carnegie Mellon University, O. G. Shivers, Carnegie Mellon University,
A. Unruh, Stanford University, and G. R. Yost, Carnegie Mellon University

Abstract

Soar is an architecture for intelligence that integrates learning into all of its problem-solving behavior. The sole learning mechanism, chunking, has been studied experimentally in a broad range of tasks and situations. This paper summarizes the research on chunking in Soar, covering the effects of chunking in different tasks, task-independent applications of chunking and our theoretical analyses of effects and limits of chunking. We discuss what and when Soar has been able to learn so far. The results demonstrate that the variety of learning in Soar arises from variety in problem solving, rather than from variety in architectural mechanisms.

1. Introduction

Soar is an architecture that is to be capable of general intelligence, displaying abilities in areas such as problem-solving, planning, diagnosis, learning, etc. (Laird, Rosenbloom, & Newell, in press). The architecture contains a single learning mechanism, called chunking, which saves the results of processing for application to future similar situations (Laird, Rosenbloom & Newell, 1984, Laird, Rosenbloom & Newell, 1986a). Chunking was first incorporated into Soar in early 1984 as a particular learning mechanism we had been exploring in the context of human practice.[1] Almost immediately thereafter chunking proved itself capable of several different types of learning (Laird, Rosenbloom & Newell, 1984, Rosenbloom, Laird, McDermott, Newell & Orciuch, 1985). At first tentatively, but then with increasing commitment, we adopted the hypothesis that chunking was a sufficient mechanism for all learning.

We have not engaged in a deliberate attempt to test this hypothesis in any definitive way. There currently exists no taxonomy of learning situations or processes that is well-founded enough to support such an endeavor, although partial classifications abound, such as learning by experience, learning by example, and learning by instruction. Instead we have proceeded to explore the operation of Soar in many different directions, observing the way learning entered into the various tasks and mechanisms studied.

The variety of learning exhibited by Soar seems extensive enough by now to make a listing of all the various

[1]More details may be found in the preface of (Laird, Rosenbloom & Newell, 1986b).

this problem go through four fairly well-defined stages in the rules used for making predictions. Initially, they base their predictions only on the weights. Then they consider the distances in order to break ties on the weights, then they break ties by combining the weights and distances in some fashion, and finally they use the weights and distances to compute the correct torques. We are investigating the hypothesis that chunking can be used to model this gradual increase in sophistication in solving this task. The Soar model of the balance beam task maps development onto a shift in the problem spaces used by the subject. Initially Soar has a set of attributes about the domain (such as weight and distance). However, it does not know exactly which attributes are relevant, or how to connect the attributes to the operators that predict which side of the balance beam will go down. Soar makes predictions, some of which are shown to be wrong by observation, and uses the results as feedback to create new problem spaces that make better predictions.

Seibel's task: As part of an investigation into the effects of practice, we have examined a reaction-time task investigated by Seibel (1963) . The task is a psychological experiment in which the subject faces a display of ten lights, and rests his or her fingers on a set of ten buttons; during each trial, some of the lights will go on, and the subject is to press the buttons corresponding to the lights that are on as quickly as possible. The reaction time data for this task obey a power law, and earlier work showed that the power law could also be obtained from a computer simulation of the process based on chunking (Rosenbloom & Newell, 1983). Soar's chunking mechanism has a mode in which it only builds chunks for terminal subgoals, which we call *bottom-up chunking*, and which can account for the effects of practice under certain assumptions. In recent work, we have replicated the power law effect for the Seibel task in Soar, assuming that reaction times correspond linearly to number of decision cycles to solution.

R1-Soar: R1-Soar was the first demonstration that Soar can combine general problem-solving methods with large amounts of domain knowledge to solve complex problems (Rosenbloom, Laird, McDermott, Newell & Orciuch, 1985). The system had about one-fourth of the functionality of R1, an expert system for configuring Vaxes. Because the chunking mechanism caches subgoal results during knowledge-intensive as well as knowledge-lean problem-solving, we were able to perform several experiments on the effects of learning on performance for the R1-Soar task. The knowledge gained by chunking was used in R1-Soar for search control and operator implementation. Chunking reduces the number of decision cycles required to find a solution even when the system already has some search control knowledge (a factor of 3.6 for the version without search control, a factor of 2.2 for partial search control, and a factor of 1.7 for full search control). We refer to this reduction as *within-trial transfer*, because the system acquires knowledge early on that it may apply to solve later subproblems within the same trial. As would be expected from an experience-learner, knowledge transferred across trials was also substantial, yielding a solution length reduction of between a factor of 20 and 200 in subsequent runs of R1-Soar on the same task.

Cypress-Soar: Another knowledge-intensive task to which we have applied Soar is algorithm design (Steier, 1987). Again, we took the approach of reimplementing a previously built system; in this case, Cypress, the semi-automatic algorithm designer built by Smith (1985) . We set about replicating the performance of Cypress in the design of three sorting algorithms: insertion-sort, quicksort, and mergesort. Cypress makes design choices for these algorithms by instantiating a functional programming template for divide-and-conquer algorithms, and propagating constraints resulting from these instantiations by a special method of deduction known as antecedent derivation. We focused only on the methods for making the design choices in Cypress-Soar, and chose not to reimplement the antecedent derivation engine. As in R1-Soar, Cypress-Soar was able to successfully perform its tasks with varying amounts of search control knowledge. Using chunking, Cypress-Soar acquires both search control and operator implementation knowledge. The effect of within-trial transfer in one run with minimal search control was to reduce the number of decision cycles by almost 70%. Across-trial transfer on design of the same algorithm gave reductions of 90-97%. The operator implementation chunks transferred across designs of different algorithms, reducing the number of decision cycles for subsequent designs by as much as 20%. We expect the 20% across-task transfer figure would have been higher had we implemented a Soar-based deduction engine.

3.2. Task-independent applications of chunking

Strategy acquisition: In the Soar implementation of tic-tac-toe, the initial knowledge for the task only included the rules for legally playing the game. By depth-first search through the space of possible moves, Soar is not only able to play tic-tac-toe, but also to learn strategies for winning. The strategies are encoded as search control chunks, and are learned without special processing beyond the chunking mechanism.

Task acquisition: Currently, the standard way for Soar to acquire new tasks is for a "Soarware engineer" to analyze the task and write a set of productions implementing the problem spaces necessary for performing the task. Since Soar is intended to function autonomously, we need to define mechanisms for Soar to acquire new task spaces on its own. Our initial approach used a problem space for building problem spaces: one that would compile task descriptions directly into productions through deliberate processing (i.e., not through chunking). We have now moved to an approach that uses the combined learning and problem solving abilities of Soar. Task acquisition is implemented by a two-phase process. The first phase is *communication* in which Soar comprehends the input and stores it internally. The second phase is *interpretation* in which Soar interpretively executes the task description, building chunks which directly implement the task. In the communication phase, the task descriptions are input in pseudo-natural language in terms of operators, desired, initial, and illegal states. These descriptions are understood by a series of word-at-a-time comprehension operators, in a method based on Steven Small's (1980) Word Expert Parser. The next phase evokes a problem space for interpretation in the event of impasses resulting from attempts to execute the task. After resolving the interpretation impasses, Soar will build task-implementation chunks, which turn out to be nearly identical to the ones that a programmer would have written by hand to directly implement the task. This two-phase process has now enabled Soar to acquire the eight-puzzle and missionaries-and-cannibals task spaces.

Data chunking: The evidence accumulated to date demonstrates that chunking can speed up Soar's performance. However, until recently there has been no evidence that chunking could also be used to acquire new factual knowledge, and doubts have been raised as to whether it is possible for such an experience-based learning mechanism to do so. Our work on data chunking is an attempt to demonstrate how chunking can be used to acquire new knowledge. In the initial phase we are focusing on two simple memory tasks: learning to recognize, and learning to recall new objects (Rosenbloom, Laird & Newell, 1987). Recognition involves determining whether the system has ever seen the object before. Recall involves the ability to generate a representation of the new object on demand. For recognition, a chunk must be learned which contains a representation of the new object in its conditions, while for recall, the object must be represented in the actions. For both tasks, the general approach we have taken is to understand what type of problem solving is required in order to achieve the appropriate chunks. For recall, the more complex of the two, performance is based on a generation problem space in which any potential object can be generated. This is a constructive space in which new objects are built up out of known pieces. When the system receives an object that it should learn to recall, it generates a representation of it, thus learning a chunk which can generate it in the future. At recall time, Soar's reflective ability is used to distinguish between the objects that it has learned to generate and those that it could potentially generate; that is, when all of the chunks have fired, an impasse occurs which the system uses as a signal that recall should be terminated. We are currently working on two more advanced memory tasks: cued recall and paired-associate recall. In cued recall, the recall of an item is conditioned on a set of discriminating cues that can be used as the basis for building a discrimination network for retrieval. This permits retrieval of an object by partial specifications and is being used as a building block in an

implementation of an EPAM-like system (Feigenbaum & Simon, 1984) for paired-associate recall.

Macro-operators: Macro-operators have been used in a variety of systems since STRIPS (Fikes, Hart & Nilsson, 1972) to improve the performance of problem-solving systems. A macro-operator is a sequence of operators that can be treated as a single operator. Korf (1983) demonstrated that it is possible to define a table of macro-operators showing how to transform all legal states into desired states if the problem is *serially decomposable*, that is, the subgoals of the problem can be ordered so that they only depend on preceding subgoals. We have replicated Korf's results in Soar by using two problem spaces: one for the domain operations of the "conventional" task decomposition, and one for operators corresponding to serially-decomposable goals (Laird, Rosenbloom, & Newell, 1986a, Laird, *et al.*, in press). The sets of search control chunks learned in the process of implementing each operator in the second space correspond to macro-operators. In a situation identical to Korf's macro-operator learner, in which there is no transfer between macro-operators, it would take 230 productions to encode the entire macro table (35 macro-operators) this way. However, Soar needs only 61 productions to encode the table because of the implicit generalization performed by chunking. The fine-grained encoding of the macro-operators as sets of chunks and the choice of an appropriate task representation makes this possible. More specifically, the generality arises because the operators are indifferent to the tiles not yet in place, and the chunks are invariant with respect to reflection, rotation, translation, and position within the solution path. In fact, so much transfer occurs with this representation that Soar can learn the entire macro table for the eight-puzzle in only three trials (given the correct set of problems).

Constraint compilation: Many tasks that have been studied in artificial intelligence can be viewed as constraint satisfaction problems (Steele, 1980). Examples are cryptarithmetic, eight-queens, and the Waltz (1975) line labeling task. If the satisfaction of a constraint is embodied in the application of an operator, then formulating constraint satisfaction as problem space search in Soar is not difficult. When a consistent assignment of values to all variables has been found, then the problem has been solved. This formulation leads to an interesting application for chunking: networks of constraints can be satisfied by subgoaling in such a way that chunks can be learned for the efficient satisfaction of a constraint network, or *macro-constraint*. We have demonstrated this form of constraint compilation for small networks of digital logic elements, with the chunks being exactly the productions that would be built by hand to compile the macro-constraint.

Learning from outside guidance: We have built a system in Soar that learns search control knowledge using outside guidance in the domain of solving simple algebraic equations in one unknown (Golding, Rosenbloom & Laird, 1987). The system has a set of algebraic operators, such as *add* and *commute*, and must decide which of these operators to apply and how to instantiate them at each step of the solution. When the system has insufficient knowledge to select an operator from a set of candidates, it asks for help, which can be given to it in one of two forms: either it can accept direct advice, or it can accept a simpler problem to solve that will illustrate the principle it is to use in solving the harder problem. If direct advice is given, the system will first verify that the advice is correct, thus protecting itself from erroneous advice. During the verification, Soar builds chunks containing the relevant search control knowledge for future use. This particular type of learning also occurs in the learning apprentice systems, where it is called *verification-based learning* (VBL) (Mahadevan, 1985). If the advice takes the form of a simpler problem to solve, Soar attempts to do so, either by brute-force search, or by accepting direct advice about the simpler problem. In the process, Soar will build chunks that summarize the solution. The chunks will transfer directly to the harder problem, provided the problem selected was appropriate.

Abstraction planning: Another kind of problem-solving behavior that has traditionally been obtained in AI systems through special-purpose mechanisms is abstraction planning (as in ABSTRIPS, Sacerdoti, 1974).

Chunking in Soar has already been used to achieve the effects of non-abstraction planning: ties in operator selection impasses are often resolved by lookahead, and the chunks formed during the lookahead apply to reduce effort in problem-solving based on the result of the lookahead. One way to make planning even more cost-effective is to use abstracted problem spaces for the lookahead search (Unruh, Rosenbloom & Laird, 1987). Abstraction of problem spaces in Soar is not based on the explicit manipulation of a declarative specification of operators. Rather, the abstraction is a consequence of ignoring parts of the problem space as needed during problem-solving. The particular abstractions possible are determined by the manner in which the problem space has been decomposed into independent subcomponents such as initial state creation, operator precondition testing, operator implementation, state evaluation and goal testing. When searching in an abstracted problem space, only the productions implementing the operations that are part of the abstracted space fire, and the chunks summarizing the processing in the abstracted search transfer to the non-abstracted space. The abstracted problem-solving has paid attention to a subset of what it would normally examine. Therefore, the chunks learned can be more general than those learned normally, and the appropriateness of the abstraction determines how useful the chunks are. This was tested in R1-Soar with several abstractions: removing the backplane-box operator, removing the modules-backplane operator, and removing the state information about module width. Two abstractions together saved 75% of the decision cycles, learning alone saved 60%, and learning together with the two abstractions saved 77%. Issues currently being investigated in this work are the automatic creation and selection of appropriate abstractions, and the use of multiple levels of abstractions.

Explanation-based generalization (EBG): Using the framework for EBG described by Mitchell, Keller and Kedar-Cabelli (1986), we have shown that chunking in Soar implements a variant of explanation-based generalization (Rosenbloom & Laird, 1986). The domain theory is implemented in some Soar problem space, and an explanation is the trace of the problem-solving in this space. Chunking uses this trace in the same way as goal regression uses a proof or explanation. The aim of regression is to describe the goal concept in terms of operational predicates, where chunking produces a description of the concept in terms of predicates on attributes of the context that existed prior to the impasse. The condition-finding algorithm in Soar's chunking is essentially the same as regression in EBG, except it uses instantiated goals and rules rather than parametrized versions. The mapping works not only for operator implementation, but also for learning search control knowledge, where the goal concept is the preference for a selection of an operator. No extra mechanisms are needed to produce the results of EBG with chunking; the problem solving that is analogous to EBG's explanation is exactly the same as is used by Soar in other problem solving. To test this mapping, we took several examples in (Mitchell, *et al.*, 1986) and implemented problem spaces in Soar that acquired nearly identical concepts to the ones acquired by "conventional" EBG algorithms.

3.3. Theoretical analysis of chunking

Sources of overgeneralization in Soar: Soar exhibits overgeneralization while chunking in several instances. We now believe that we understand most, if not all, sources of overgeneralization in Soar (Laird, Rosenbloom & Newell, 1986c). Our investigation was based on the theory of chunking as *knowledge compilation*, that is, the conversion of knowledge from one form to another so that it may be used in similar situations but more efficiently than before. For this conversion to be correct and avoid overgeneralization in Soar, all of the relevant aspects of the pre-impasse situation that lead to the creation of results must be included as conditions of a chunk. This is easy as long as the results of a subgoal depend only on the productions that fired during the subgoal. However, in some subgoals, the appropriate result (such as success or failure) is most easily detected by testing some feature of the problem solving process, such as the available operators are exhausted and there nothing more to do. For example, if the highest number of a set is to be found, and each step of the problem solving involves comparing one of the numbers to the highest found so far, the problem is finished when no other numbers are left to be compared. In Soar, detecting that there is nothing more to do

means testing that no more productions can fire for the current situation, which in Soar becomes testing for the existence of an impasse. For chunking to be correct, it must determine why no more productions fired and include these reasons in the chunk. Although this may be logically possible, it is computationally intractable and the resulting chunks can be less efficient than the original problem solving. Therefore, these conditions are not included and the resulting chunks are overgeneral. This problem arises whenever the problem solving for a goal is dependent on its own process state (meta-information) which is made available in Soar though the impasses. Because of these problems, we are currently investigating a variety of methods to recover from overgeneralization in these cases when it cannot be prevented.

Chunking and meta-levels: We have also analyzed Soar in terms of concepts such as meta-levels, introspection and reflection (Rosenbloom, Laird & Newell, 1986). Several meta-levels were identified, and chunking was cast as a means for shifting knowledge from the problem-space meta-level, where its application is slow and deliberate, to the production meta-level, where its application is fast and automatic. This production meta-level acts as a long-term cache for the results of problem solving at the higher problem space level, and improves the efficiency of the problem solver by reducing the need to climb the hierarchy of meta-levels as often.

4. Soar As A Learning Architecture

Table 2 lists some questions that might be asked about a particular learning architecture in order to assess its abilities and compare it to others. This section attempts to answer these questions with regard to Soar.

Table 2: Questions to ask about learning architectures

1. When can the architecture learn?
- On what kinds of tasks?
- On what occasions during performance of the tasks?

2. From what can the architecture learn?
- Internal sources of knowledge?
- External sources of knowledge?

3. What can the architecture learn?
- What kinds of procedural knowledge?
- What kinds of declarative knowledge?

4. When can the architecture apply learned knowledge?
- Within the same trial?
- Across different trials?
- Across different tasks?

4.1. When can Soar learn?

When working on any type of tasks: These tasks span the range from simple and knowledge-lean to complex and knowledge-intensive, from routine to design and discovery. Soar has exhibited interesting learning on instances of all of these, and is theoretically capable of learning on any task it can perform. The simple tasks include the eight-puzzle, tic-tac-toe, towers of Hanoi, missionaries-and-cannibals, solving simple algebraic equations in one unknown, satisfying logical constraint networks, solving logical syllogisms, balance beam, simple blocks world, eight-queens, monkey-and-bananas, and the water jug problems. The complex tasks include computer configuration, algorithm design, and applying sociological theories. Routine tasks include paired-associate learning and the Seibel task. One discovery task is algorithm design.

Whenever a result is generated: The chunking mechanism is the same for results at any level in the goal hierarchy, and is invoked independently of the success or failure of the goal. For example, R1-Soar builds a chunk for the top-level goal during the successful configuration of a VAX backplane. Learning from failure occurs during the missionaries-and-cannibals runs; if a move is made that results in there being more cannibals than missionaries on a bank or in the boat (an illegal state), Soar learns not to make that move again.

4.2. From what can Soar learn?

Internal sources of knowledge: The normal source of learning for Soar is its own problem solving experience, since it usually functions autonomously. For example, Soar has learned to solve the missionaries-and-cannibals problems by brute-force search, and to satisfy macro-constraints based on solutions to constraint networks found previously. The chunking mechanism for all this is uniform.

External sources of knowledge: Several of the research efforts described earlier illustrate how Soar can learn from knowledge provided to it externally. In the algebraic domain, the user can supply direct advice on which of several operators to apply to an equation. Soar can also obtain the same knowledge if the user supplies it with a similar, simpler problem to solve. The chunks built after solving the simpler problem will transfer to provide the search control for the original problem. Two other sources of external knowledge now being tapped by Soar are pseudo-natural language descriptions of task problem spaces and declarative facts.

4.3. What can Soar learn?

Procedural knowledge: A step in a procedure in Soar corresponds to the replacement of a single element in a goal context. A goal context consists of a goal, problem space, state and operator. We can analyze what procedural knowledge Soar can learn, and has been demonstrated to learn, by examining what aspects of this context replacement are subject to modification by chunking. Soar learns both control ("when" or "which" knowledge) and implementation ("how" knowledge) for each of the components of a context (except goals).

- **Goals:** The set of possible goal types forms a complete set, fixed by the architecture; goal types are determined by the decision cycle in response to impasses in a manner also fixed by the architecture. Consequently, the effect of chunking on the goal element of a context is limited to avoiding the need for subgoaling.
- **Problem spaces:** A problem space consists of a set of possible states and operators, and the architecture does not impose any restrictions on what constitutes a legal set. Therefore, Soar must generate candidate problem spaces, and select from among the candidates. For generation, Soar can either acquire problem spaces from external descriptions, or can modify problem spaces it already has to obtain new ones. The work on task acquisition has shown Soar to be capable of the former. For the latter, problem spaces have been abstracted from old ones by omitting operators in the work on abstraction planning, and Soar builds new problem spaces during the balance beam task as well. For learning problem space selection knowledge, we have only simple examples, in the context of task acquisition and the balance beam task.
- **States:** The selection of a problem space imposes certain constraints on the creation and selection of states. Our main experience to date with acquiring initial state creation and selection knowledge has been through task acquisition. For creation of states other than initial ones, the processing, and thus the knowledge to acquire by chunking, is primarily determined by the operator selected for application to the previous state. Much of the transfer due to chunking comes from learning how to create states, or implement operators, without subgoaling. For example, in Cypress-Soar, an operator in the top-level space creates a divide-and-conquer sorting algorithm to satisfy a sorting specification. The first time through the problem, this operator will be implemented by subgoaling into a special space for designing divide-and-conquer algorithms, and a chunk will be built that creates an algorithm such as quicksort for that specification. On subsequent trials, that chunk will

fire to create the new state with the algorithm directly.

- **Operators:** All of the R1-Soar operator implementation work (and many other tasks) demonstrates that Soar creates operators from subgoals. Operator creation is also part of task acquisition, and an equivalent effect was achieved in learning macro tables for the eight-puzzle and Tower of Hanoi. For operator selection knowledge, examples abound in the toy problems, in R1-Soar and Cypress-Soar and in the research on learning from outside guidance and abstraction planning. Most operator selection, or search control, chunks result from summarizing the processing in lookahead search, where it is learned that a particular operator application will be on the path to a solution.

Declarative knowledge: The work on recognition, free recall, and cued recall shows that Soar can learn declarative knowledge.

4.4. When can Soar apply learned knowledge?

Within the same trial: Application of knowledge learned while solving a problem to improve performance in a later stage of solving the same problem is called *within-trial transfer*. This may be measured by comparing the number of decision cycles Soar takes to solve a given problem without learning against how many it takes with learning. The reduction can be substantial: a factor of 2 for some runs of the eight puzzle, over a factor of 2 for Cypress-Soar, and a factor of 3 for R1-Soar. The transfer occurs because of the presence of repeated subgoals during a trial for these problems.

Across different trials: When the system builds chunks for every terminated goal, the only effort required for solving a problem on the second and subsequent trials is the processing demand by solving the problem with no search. A chunk in R1-Soar reduces the number of decision cycles in a solution of the same problem on subsequent trials by a factor of 20 to 200 in some cases. If the system is learning bottom-up, where chunks are only built for terminal goals, the processing that is repeated most often gets chunked first. Eventually, the results of bottom-up chunking will be the same as those of all-goals chunking. Work on the Seibel reaction-time task has shown that the across-trial transfer with bottom-up chunking leads to a power law effect on performance.

Across different tasks: Across-task transfer occurs in Soar when similar subgoals show up in different tasks; this happened in Cypress-Soar, in the Seibel task, in several of the toy problems, and to some extent in R1-Soar. For example, in the divide-and-conquer formulation of all the sorting algorithms synthesized by Cypress-Soar, a sub-algorithm must be created to sort lists of length zero or one. Once this sub-algorithm, which merely returns its input, was designed for one algorithm, the chunk built during that design could fire on the design of other algorithms. Once one of the three algorithms had been designed, such chunks reduced by 10 to 20 percent the decision cycles needed to design the other two algorithms.

4.5. What variety of learning has Soar not yet exhibited?

In order to support the claim that chunking is a truly general learning mechanism, it must be studied further in a number of areas. A partial list of these areas follows:

- **Very large problems:** We have not yet tested learning in Soar in large (greater than 10^3 rules) systems for solving very difficult problems, though this may soon be possible as a consequence of work in progress on Designer-Soar (for algorithm design tasks) and Neomycin-Soar (for medical diagnosis tasks).
- **Complex analogies:** In learning from outside guidance provided as a simpler problem to solve, Soar performs a rudimentary form of analogy, which requires that the chunks transfer directly. Soar is not yet capable of more complex analogical reasoning involving deliberate processing to adapt the solution to the simpler problem.

- **Similarity-based generalization:** Soar does not yet use multiple examples in order to induce new concepts. Early work on a version-space learner within Soar did not make use of chunking.
- **Representation shifts:** In the category of problem space creation, we have yet to demonstrate the creation of "completely new" representations in a task. While there is a moderate shift of representation in the balance beam task, the set of possible problem space modifications must be somehow "designed in" beforehand.

5. Conclusion

We have described how Soar successfully learns in a wide range of tasks and situations. Most of these results could previously only be obtained from special-purpose learning mechanisms. The variety of learning possible in Soar does not arise from collecting all such mechanisms into one system. Rather, they have been produced with a single mechanism, which is general enough to apply to all problem-solving behavior. Thus the diversity of problem-solving of which Soar is capable is directly responsible for the varieties of learning in Soar.

Acknowledgements

Brian Milnes made comments on an earlier draft of this paper. This research was sponsored by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 3597, monitored by the Air Force Avionics Laboratory under contracts F33615-81-K-1539 and N00039-83-C-0136, and partially by the National Science Foundation under grant DCR-8412139. The research was also partially supported by graduate fellowships from Bell Laboratories to Andrew Golding, and from the National Science Foundation to Gregg Yost. Additional partial support was provided by the Sloan Foundation, and some computing support was supplied by the SUMEX-AIM facility (NIH grant number RR-00785). 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 National Science Foundation, the Sloan Foundation, the National Institute of Health, or the U.S. Government.

References

Feigenbaum, E. A. and Simon, H. A. (1984). EPAM-like models of recognition and learning. *Cognitive Science, 8*, 305-336.

Fikes, R., Hart, P., and Nilsson, N. (1972). Learning and executing generalized robot plans. *Artificial Intelligence, 3*, 251-288.

Forgy, C. L. (1981). *OPS5 user's manual* (Tech.Rep. CMU-CS-81-135). Pittsburgh, PA: Carnegie-Mellon University, Computer Science Department, .

Golding, A., Rosenbloom, P. S., and Laird, J. E. (1987). Learning general search control from outside guidance. In *Proceedings of the Tenth International Conference on Artificial Intelligence* (to appear). Milano, Italy: Morgan Kaufmann.

Johnson-Laird, P. N. (1983). *Mental models: Towards a cognitive science of language, inference, and consciousness.* Cambridge, MA: Harvard University Press.

Korf, R. E. (1983). *Learning to solve problems by searching for macro-operators.* Doctoral dissertation, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA.

Laird, J. E. *Soar user's manual* (Tech. Rep. ISL-15). Palo Alto, CA: Intelligent Systems Laboratory, Xerox Corporation.

Laird, J. E., Newell, A., and Rosenbloom, P. S. (in press). Soar: An architecture for general intelligence. *Artificial Intelligence.*

Laird, J. E., Rosenbloom, P. S., and Newell, A. (1984). Towards chunking as a general learning mechanism. In *Proceedings of the Fourth National Conference on Artificial Intelligence* (pp. 188-192). Austin, TX:

Morgan Kaufmann.

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

Laird, J. E., Rosenbloom, P. S., and Newell, (1986b). A. *Universal subgoaling and chunking: The automatic generation and learning' of goal hierarchies.* Hingham, MA: Kluwer Academic Publishing.

Laird, J. E., Rosenbloom, P. S., and Newell, A. (1986c). Overgeneralization during knowledge compilation in Soar. In *Proceedings of the Workshop on Knowledge Compilation* (pp. 46-57). Otter Crest, OR.

Mahadevan, S. (1985) Verification-based learning: A generalization strategy for inferring problem reduction methods. In *Proceedings of the Ninth International Conference on Artificial Intelligence* (pp. 616-623). Los Angeles, CA: Morgan Kaufmann.

Mitchell, T. M., Keller, R. M., and Kedar-Cabelli, S. T. (1986). Explanation-based generalization: A unifying view. *Machine Learning, 1*, 47-80.

Rosenbloom, P. S., and Laird, J. E. (1986). Mapping explanation-based generalization onto Soar. In *Proceedings of the Fifth National Conference on Artificial Intelligence* (pp. 561-567). Philadelphia, PA: Morgan Kaufmann.

Rosenbloom, P. S. and Newell, A. (1983). The chunking of goal hierarchies: A generalized model of practice. In *Proceedings of the 1983 International Machine Learning Workshop* (pp. 183-197). Urbana-Champaign, IL.

Rosenbloom, P. S., Laird, J. E., and Newell, A. (1986). Meta-levels in Soar. In *Preprints of the Workshop on Meta-level Architectures and Reflection.* Sardinia.

Rosenbloom, P. S., Laird, J. E. and Newell, A. (1987). Knowledge-level learning in Soar. In *Proceedings of the Sixth National Conference on Artificial Intelligence* (to appear). Seattle, WA: Morgan Kaufmann.

Rosenbloom, P. S., Laird, J. E., McDermott, J., Newell, A., and Orciuch, E. (1985). R1-Soar: An experiment in knowledge-intensive programming in a problem-solving architecture. *IEEE Transactions on Pattern Analysis and Machine Intelligence, 7*, 561-569.

Sacerdoti, E. D. (1974). Planning in a hierarchy of abstraction spaces. *Artificial Intelligence, 5*, 115-135.

Seibel, R. (1963). Discrimination reaction time for a 1,023-alternative task. *Journal of Experimental Psychology, 66*, 215-226.

Siegler, R. S. (1978) The origin of scientific reasoning. In R. S. Siegler (Ed.), *Children's thinking: What develops?* Hillsdale, N.J.: Lawrence Erlbaum and Associates.

Small, S. (1980) *Word expert parsing: A distributed theory of word-based natural language understanding.* Doctoral dissertation, Department of Computer Science, University of Maryland, College Park, MD.

Smith, D. R. (1985). Top-down synthesis of divide-and-conquer algorithms. *Artificial Intelligence, 27*, 43-96.

Steele, G. L., Jr. (1980). *The Definition and Implementation of a Computer Programming Language Based on Constraints.* Doctoral dissertation, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, MA.

Steier, D. M. (1987). Cypress-Soar: A case study in search and learning in algorithm design. In *Proceedings of the Tenth International Conference on Artificial Intelligence* (to appear). Milano, Italy: Morgan Kaufmann.

Unruh, A., Rosenbloom, P. S., and Laird, J. E. (1987). *Dynamic abstraction problem solving in Soar.* In preparation.

Waltz, D. L. (1975). Understanding line drawings of scenes with shadows. In P. Winston (Ed.), *The Psychology of Computer Vision.* New York, NY: McGraw-Hill.