Learning General Search Control from Outside Guidance*

Andrew Golding and Paul S. Rosenbloom
Computer Science Department
Stanford University
Stanford, CA 94305

John E. Laird
EECS Department
University of Michigan
Ann Arbor, MI 48109

Abstract

The system presented here shows how Soar, an architecture for general problem solving and learning, can acquire general search-control knowledge from outside guidance. The guidance can be either direct advice about what the system should do, or a problem that illustrates a relevant idea. The system makes use of the guidance by first formulating an appropriate goal for itself. In the process of achieving this goal, it learns general search-control chunks. In the case of learning from direct advice, the goal is to verify that the advice is correct. The verification allows the system to obtain general conditions of applicability of the advice, and to protect itself from erroneous advice. The system learns from illustrative problems by setting the goal of solving the problem provided. It can then transfer the lessons it learns along the way to its original problem. This transfer constitutes a rudimentary form of analogy.

I. Introduction

Chunking in Soar has been proposed as a general learning mechanism [Laird *et a*l., 1986]. In previous work, it has been shown to learn search control, operator implementations, macro-operators, and other kinds of knowledge, in tasks ranging from search-based puzzles to expert systems. Up to now, though, chunking has not been shown to acquire knowledge from the outside world. The only time Soar learns anything from outside is when the user first defines a task; but this is currently done by typing in a set of productions, not by chunking. The objective of the research reported here is to show that chunking can in fact learn from interactions with the outside world that take place during problem solving.

Two particular styles of interaction are investigated in the present work. Both come into play when Soar has to choose among several courses of action. In the first, the advisor tells Soar directly which alternative to select. In the second, the advisor supplies a problem within Soar's grasp that illustrates what to do. Both styles of interaction teach Soar search-control knowledge.

In the next section, the basics of the Soar architecture are laid out. A Soar system that implements the two styles of interaction mentioned above is then described. Following that is a discussion of related work and directions for future research. The final section summarizes the contributions of this work.

II. The Soar Architecture

Soar [Laird *et al.,* 1987] is an architecture for general cognition. In Soar, all goal-oriented behavior is cast as search in a problem space. Search normally proceeds by selecting an operator from the problem space and applying it to the current state to produce a new state. The search terminates if a state is reached that satisfies the goal test. All elements of the search task - operators, problem spaces, goal tests, etc. — are implemented by productions.

Various difficulties can arise in the course of problem solving. An *operator-tie impasse* results when Soar is unable to decide which operator to apply to the current state. An *operator no-change impasse* occurs when Soar has selected an operator, but does not know how to apply it to the state. When Soar encounters any kind of impasse, a subgoal is generated automatically to resolve it. This subgoal, like the original goal, is solved by search in a problem space — thus Soar can bring its full problem-solving capabilities to bear on the task of resolving the impasse.

Chunking is the learning mechanism of Soar. It summarizes the processing of a subgoal in a *chunk.* The chunk is a production whose conditions are the inputs of the subgoal, and whose actions are the final results of the subgoal. Intuitively, the inputs of a subgoal are those features of the pre-subgoal situation upon which the results depend. Because certain features are omitted from the chunk namely those that the results do not depend on — the chunk attains an appropriate degree of generality.

Once a chunk is learned for a subgoal, Soar can apply it in relevant situations in the future. This saves the effort of going into another subgoal to rederive the result.

III. Design of the System

In this section, a Soar system that learns search control from outside guidance is presented. The performance task of the system is described first, and then the strategy for learning from interactions with the outside world.

*This research was sponsored by the Defense Advanced Research Projects Agency (DOD) under contract N00039-86-C-0133, by the Sloan Foundation, and by a Bell Laboratories graduate fellowship to Andrew Golding. 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, Bell Laboratories, or the National Institute of Health. The authors are grateful to Allen Newell and Haym Hirsh for comments on earlier drafts of this paper.

A. The Performance Task

The system works in the domain of algebraic equations containing a single occurrence of an unknown, such as

$$(a + x)/(b + c) = d + e. \quad (1)$$

The unknown here is x, and the solution is

$$x = (d + e)-(b + c)-a. \quad (2)$$

To derive this solution, the system goes into its *equations* problem space and applies an appropriate sequence of transformations to the equation. Transformations are effected by *isolate* and *commute* operators. Together, these two types form a minimal sufficient set for solving any problem in the task domain of the system.

Isolate operators transfer top-level terms from one side of the equation to the other. To illustrate, two *isolate* operators are applicable to equation 1. One multiplies the equation by the term *(b* + c), yielding

$$a + x = (d + e)-(b + c); \quad (3)$$

the other subtracts e from both sides and produces

$$(a+x)/(b+c) - e = d. \quad (4)$$

The reason the latter subtracts e from both sides, and not *d,* is that *isolate* operators always transfer *right-hand* arguments. *Isolate* operators can thus be characterized by two parameters: the operation they perform, and the side of the equation the transferred term starts out on. There are five possible operations: *add, subtract, multiply, divide,* and *unary-minus;* the side of the equation is either *left* or *right.* The operators above are *isolate(multiply, left)* and *isolate(subtract, right),* respectively.

There are just two *commute* operators, one for each side of the equation. They swap the arguments of the top-level operation on their side, provided the operation is abelian. *Commute (right)* applies to equation 1, giving

$$(a + x)/(b + c) = e + d. \quad (5)$$

B. The Learning Strategy

The system's strategy for learning is to ask for guidance when needed, and to translate that guidance into a form that can be used directly in solving equations. The general procedure for doing this is to formulate an appropriate goal and then achieve it, thereby learning chunks that influence performance on the original equations task. The instantiation of the general procedure depends on the form of guidance provided. The most direct form is where the advisor tells the system which of its operators to apply. Alternatively, the advisor could suggest an easier problem whose solution illustrates what to do in the original problem. Less direct still, the advisor could refer the system to a textbook, or offer other equally hostile assistance.

The current system accepts both of the simpler forms of help mentioned above — direct advice and illustrative problems. In the following sections, the general procedure is instantiated and illustrated for these forms of help.

1. Learning from Direct Advice

In the course of solving its equations, the system is likely to run into situations where it does not know which operator to apply next. This condition is signalled in Soar as an operator-tie impasse in the *equations* space. Soar's default behavior is to go into a *selection* problem space and proceed to evaluate each operator involved in the tie in an arbitrary order, until a correct operator is found.

At this point, there is an opportunity for the system to benefit from outside guidance. Rather than evaluate operators randomly, the system enters an *advise* problem space, where it displays all of the operators, and asks the advisor to pick one. The advisor's choice is evaluated first, in the hopes that it will be correct, allowing the evaluation process to be cut short. Normally, the system will be unable to evaluate the advisor's choice by inspection; thus it sets up a subgoal to do the evaluation. In the subgoal, it goes into another *equations* space and applies the advisor's choice to the current equation. If this leads to a solution, the operator is accepted as correct.

Phrased in terms of the general procedure given above, the goal that the system sets for itself is to evaluate the correctness of the advisor's guidance — in this case, the system resembles a traditional learning apprentice; this point is taken up in section IV.A. A more trusting system would simply have applied the suggested operator to its equation. However, by verifying that the recommended operator leads to a solution, the system paves the way for the learning of general conditions of applicability of the operator. This is done by the chunking mechanism, which retains those features of the original equation that were needed in the verification, and discards the rest. Moreover, if the system is given a wrong** operator, its verification will fail, and thus it will gracefully request an alternative suggestion. It even picks up valuable information from the failed verification by analyzing what went wrong; the chunks learned from this analysis give general conditions for when *not* to apply the operator.

2. Example of Direct Advice

Following is a description of how the system solves

$$ab= -c- x. \quad (6)$$

together with direct advice from outside. A graphic depiction of the problem solving appears in Figure 1.

Since the system has no prior search-control knowledge, it cannot decide which operator to apply to the initial equation. It asks for a recommendation, and the advisor gives it *isolate(add, right).* The system sets up a subgoal to evaluate the correctness of this operator. In the subgoal, it tries out the operator on its equation, yielding

$$a.b + x = -c. \quad (7)$$

Here the evaluation runs into a snag, as the system again cannot decide on an operator. It asks for help, and is told to apply *commute(left).* Accordingly, it sets a subgoal within its current evaluation subgoal to verify this advice. The first equation generated in the new subgoal is

$$x + a \bullet b = -c. \quad (8)$$

**It turns out that in the current domain, the only wrong operators are those that are inapplicable to the equation.

The final bit of guidance the system needs is that it should apply the *isolate(subtract, left)* operator. It goes down into a third nested subgoal to verify this, and obtains

$$x = -c-ab. \quad (9)$$

Having reached a solution, the system is satisfied that *isolate(subtract, left)* is correct, and Soar learns a chunk that summarizes the result. The chunk states that if the left-hand side of the equation is a binary operation with the unknown as its left argument, then the best operator to apply is the one that undoes that binary operation.

The verification of the preceding operator, *commute(left),* now goes through as well. The chunk for this subgoal pertains to equations with an abelian binary operation on the left-hand side, whose right argument is the unknown. It says to apply the *commute(left)* operator.

Finally, the first evaluation subgoal terminates, and a chunk is learned for it. This chunk requires that the equation have on its right-hand side a binary operation whose inverse is commutative, and whose right argument is the unknown. It asserts that the best operator to apply is the one that undoes the binary operation.

Having done the necessary evaluations, the system can go ahead and solve the equation. It no longer has to ask for advice, because its chunks tell it which operators to apply. These chunks may also prove useful in other problems, as demonstrated in the next two sections.

3. Learning from Illustrative Problems

Learning from an illustrative problem takes place in the same context as learning from direct advice — namely, when Soar is about to evaluate the operators involved in a tie. But now, instead of going into an *advise* problem space, the system enters another *equations* space. This instance of the space is for solving the illustrative problem.

The initial state of this space does not contain an equation. The system detects this, and goes into a *parse* problem space, where it asks the advisor for an illustrative equation. It parses the equation into its tree-structured internal representation, and attaches it to the initial state; now it is ready to attempt a solution.

To instantiate the general procedure presented earlier, the goal the system sets for itself this time is to solve the illustrative example. There are several ways for it to do so — the current system can either follow direct advice, as described above, or do an exhaustive search.

In the process of solving the illustrative problem, chunks will be learned that summarize each subgoal. Then, if the subgoals of the illustrative problem are sufficiently similar to those of the original problem, the chunks should apply directly, resolving the original operator-tie impasse. The learning strategy is thus to apply the lessons of one problem to another. This can be viewed as a rudimentary kind of analogical transfer, as discussed in section IVB.

4. Example of an Illustrative Problem

In this run, the system is again asked to solve

$$a-b = -c-x. \quad (10)$$

Figure 2 gives a pictorial representation of the run.

As before, the system hits an operator-tie impasse at the first step, but this time the advisor helps by supplying

$$r = s/y \quad (11)$$

as an illustrative problem. This problem is simpler than the original one, as it has no extraneous operations, such as a multiplication or unary minus, to distract the system. An exhaustive breadth-first search would expand 28 nodes in solving the original equation, but only 7 nodes here.

The system proceeds to solve equation 11 by brute-force search. The details are suppressed here, but the outcome is that it finds the sequence of operators *isolate (multiply, right), commute (left),* and *isolate(divide, left).* Chunks are learned for each step of this solution. These chunks are in fact identical to the chunks learned in section III.B.2; this is because equations 10 and 11 are identical in all *relevant* aspects. It follows that the chunks can be applied directly to solve the original problem.

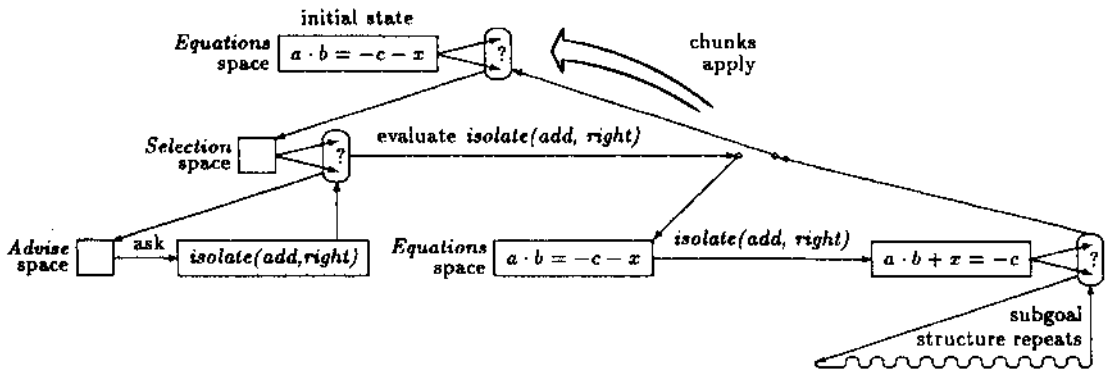

Figure 1: Subgoal structure for learning from direct advice. The levels of the diagram correspond to subgoals. Non-horizontal arrows mark the entry and exit from subgoals. Within a subgoal, a box represents a state, and a horizontal arrow stands for the application of an operator. Ovals denote operator-tie impasses. Operator no-change impasses appear as two circles separated by a gap. A wavy line symbolizes arbitrary processing.

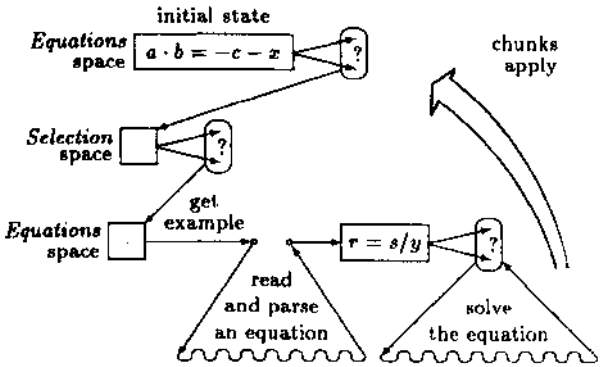

Figure 2: Subgoal structure for learning from an illustrative problem.

IV. Discussion

The system described here represents a first step toward the construction of an agent that is able to improve its behavior merely by taking in advice from the outside. Soar appears ideally suited as a research vehicle to this end, as it provides general capabilities for problem solving and learning that were not available in previous research efforts [McCarthy, 1968, Rychener, 1983]. The relatively straightforward implementation of direct advice and illustrative problems shows that the advice-taking paradigm fits naturally into Soar. Below, these methods of taking advice are compared with related work, and extensions to the straightforward implementations are proposed.

A. Direct Advice

The system is similar to a learning apprentice [Mitchell *et al.*, 1985] in its treatment of direct advice; instead of accepting it blindly, it first explains to itself why it is correct. Nevertheless, the system cannot accurately be called a learning apprentice, as it *actively* seeks advice, as opposed to passively monitoring the user's behavior. In fact, the learning-apprentice style of interaction could be considered a special case of advice-taking in which the guidance consists of a protocol of the user's problem-solving.

The limitation of direct advice is that it forces the advisor to name a particular operator; it would be desirable to allow higher-level specifications of what to do. To take the canonical example of the game of Hearts, the advisor might want to tell the system to play a card that avoids taking points, instead of spelling out exactly which card to play. To accept such indirect advice, the system would have to reduce it to a directly usable form [Mostow, 1983].

B. Illustrative Problems

The system processes an illustrative problem by applying the chunks it learns from that problem to the original one. Since it is solving the two problems in serial order, it may seem that this approach amounts to just working through a graded sequence of exercises. There are two reasons that it does not, however. First, the teacher can observe how the student fails, and take this into account in choosing a suitable illustrative problem. Second, the system is solving the illustrative problem *in service of* the original one; thus it can abandon the illustrative problem as soon as it learns enough to resolve the original impasse.

A more apt way to view the system's processing of illustrative problems is as a type of analogical transfer from the illustrative to the original problem. The trouble with this type of analogy, though, is that the generalizations are based solely on the source problem, without regard for how they will apply to the target. A more effective approach would be to establish a mapping between the two problems explicitly. This forces the system to attend to commonalities between the problems, which would then be captured in its generalizations. This is in fact just the way generalizations are constructed in GRAPES [Anderson, 1986].

V. Conclusion

The system presented here shows how Soar can acquire general search-control knowledge from outside guidance. The guidance can be either direct advice about what the system should do, or a problem that illustrates a relevant idea. The system's strategy of verifying direct advice before accepting it illustrates how Soar can extract general lessons, while protecting itself from erroneous advice. This strategy could be extended by permitting the advice to be indirect; the. system would then have to operationalize it. In applying the lessons learned from solving an illustrative problem to its original task, the system demonstrates an elementary form of analogical reasoning. This reasoning capability could be greatly improved, however, if the system were to take into consideration the target problem of the analogy as well as the source problem.

References

John R. Anderson. Knowledge compilation: the general learning mechanism. In *Machine Learning: An Artificial Intelligence Approach,* pages 289 310, Morgan Kaufmann, Los Altos, CA, 1986.

John E. Laird, Allen Newell, and Paul S. Rosenbloom. Soar: an architecture for general intelligence. *Artificial Intelligence,* 1987. In press.

John E. Laird, Paul S. Rosenbloom, and Allen Newell. Chunking in Soar: the anatomy of a general learning mechanism. *Machine Learning,* 1, 1986.

John McCarthy. Programs with common sense. In Marvin Minsky, editor, *Semantic Information Processing,* pages 403 418, MIT Press, Cambridge, MA, 1968.

T. Mitchell, S. Mahadevan, and L. Steinberg. LEAP: a learning apprentice for VLSI design. In *Proceedings of IJCAI-85,* Los Angeles, 1985.

David Jack Mostow. Machine transformation of advice into a heuristic search procedure. In *Machine Learning: An Artificial Intelligence Approach,* pages 367-404, Tioga, Palo Alto, CA, 1983.

Michael D. Rychener. The Instructible Production System: a retrospective analysis. In *Machine Learning: An Artificial Intelligence Approach,* pages 429-460, Tioga, Palo Alto, CA, 1983.