[8] Clifford Lasser. *The Essential *Lisp Manual.* Technical Report, Thinking Machines Corporation, Cambridge, MA, 1986.

[9] Daniel Paul Miranker. *TREAT: A New and Efficient Match Algorithm for AI Production Systems.* PhD Thesis, Columbia University, 1987.

[10] Allen Newell. *Scale Counts in Cognition.* Distinguished Scientific Contribution Award lecture, American Pschological Association Meeting, Washington DC, 1986.

[11] Allen Newell. *Unified Theories of Cognition.* William James Lectures, Harvard University, Cambridge MA, 1987.

[12] Kemal Oflazer. *Partitioning in Parallel Processing of Production Sytstems.* PhD Thesis, Carnegie-Mellon University, 1986.

[13] —. *The Connection Machine Parallel Instruction Set (PARIS).* Thinking Machines Corporation, Cambridge, MA. 1986.

CHAPTER 22

Recovery from Incorrect Knowledge in Soar

J. E. Laird, University of Michigan

Abstract

Incorrect knowledge can be a problem for any intelligent system. Soar is a proposal for the underlying architecture that supports intelligence. It has a single representation of long-term memory and a single learning mechanism called chunking. This paper investigates the problem of recovery from incorrect knowledge in Soar. Recovery is problematic in Soar because of the simplicity of chunking: it does not modify existing productions, nor does it analyze the long-term memory during learning. In spite of these limitations, we demonstrate a domain-independent approach to recovery from incorrect control knowledge and present extensions to this approach for recovering from all types of incorrect knowledge. The key idea is to correct decisions instead of long-term knowledge. Soar's architecture allows this corrections to occur in parallel with normal processing. This approach does not require any changes to the Soar architecture and because of Soar's uniform representations for tasks and knowledge, this approach can be used for all tasks and subtasks in Soar.

1 Introduction

Incorrect knowledge is a fact of life for any intelligent system, be it natural or artificial. There are many potential origins of incorrect knowledge: mistakes in the original coding of a knowledge-based system; errors in learning [Laird *et al.*, 1986b]; or changes in the state of the world that invalidate prior knowledge. No matter what the reason for the incorrect knowledge, an intelligent system must have the capability to overcome the effects of errors in its long-term knowledge.

The purpose of this paper is to investigate recovery from incorrect knowledge within Soar, an integrated problem solving and learning architecture for building intelligent systems [Laird *et al.*, 1987]. Its learning mechanism, called chunking, acquires productions based on problem solving in subgoals. It is a variant of explanation-based learning (EBL) [DeJong and Mooney, 1986; Mitchell *et al.*, 1986; Rosenbloom and Laird, 1986] and knowledge compilation [Anderson, 1983]. Within explanation-based learning, Rajamoney and DeJong have suggested a general experimentation approach for dealing with imperfect domain theories [Rajamoney and DeJong, 1987], both Chien and Hammond have demonstrated failure-driven schema refinement mechanisms [Chien, 1987; Hammond, 1986], while Doyle has demonstrated recovery using supporting layers of domain theories to refine inconsistent theories [Doyle, 1986]. Our work builds on these efforts, but our goal is to integrate recovery within a general problem solving and learning system so that recovery is possible for all types of incorrect knowledge and all types of tasks.

A secondary motivation for this research is to test the hypothesis that chunking is sufficient for all cognitive learning in Soar [Laird *et al.*, 1986a; Rosenbloom *et al.*, 1988; Rosenbloom *et al.*, 1987; Steier *et al.*, 1987]. The importance of this hypothesis is that it provides a simple but general theory for integrating learning and performance in all tasks. Some of the ramifications of this hypothesis are:

1. There is only a single learning mechanism.
2. All long-term knowledge is represented as productions.
3. Learning is a background process, not under control of the problem solver.
4. Long-term knowledge is only added, never forgotten, modified or replaced.

Demonstrating Soar's ability or inability to recover from incorrect knowledge is of theoretical interest because architectural assumptions prohibit most traditional correction techniques, such as modifying the conditions of a production [Langley, 1983], deleting a production, lowering the strength of a production [Holland, 1986], or masking an incorrect production through conflict resolution. If recovery from incorrect knowledge is not possible using chunking in Soar, our hypothesis will have to be abandoned.

2 Overview of Soar

In Soar, all tasks and subtasks are cast as searches in problem spaces, where operators generate new states until a desired state is achieved. All knowledge of a task—operators implementations, control knowledge, goal tests—is encoded in productions. Therefore, incorrect knowledge is encoded as an incorrect production.

Productions encode all long-term knowledge, acting as a memory, only adding elements to working. They are not the locus of deliberation or control. In contrast to Ops5 [Forgy, 1981], a typical production system, there is no conflict resolution in Soar and all productions fire in parallel until quiescence. All decisions are made by a fixed procedure based on working-memory elements called *preferences*. These decisions perform all the basic acts of

*This research was sponsored in part by grant NCC2-517 from NASA Ames.

problem solving in a problem space: selecting the current problem space, state and operator for a goal.

There are three classes of preferences: acceptability, necessity, and desirability. In the acceptability class, acceptable preferences specify those objects that are candidates for a slot, while reject preferences eliminate a candidate from consideration. Necessity preferences enforce constraints on goal achievement by specifying either that an object must be selected (require) or must not be selected (prohibit) in order that the goal be achieved. The priority of these preferences is: `(prohibit, require) > reject > accept`. Therefore, an object will only be considered if it is acceptable and not rejected and not prohibited, or if it is required and not prohibited.

Desirability preferences also control the selection of candidates, but provide only heuristic information. In other words, the necessity preferences encode knowledge to ensure the correctness of the problem solving, while the desirability preferences encode knowledge to improve the efficiency of the problem solving. Desirability preferences provide either absolute (best, indifferent, worst) or relative (better, indifferent, worse) orderings of the candidates for a slot. The desirability preferences have their own precedence ordering: `(better, worse) > best > worst > indifferent`. That is, better and worse preferences are considered first, and only those candidates not worse than some other candidate are considered by the remaining preferences. If only a single object is preferred at the end of this procedure, it is selected. If there are multiple objects that are either all indifferent, all best, or all worst, a random selection is made between them. If none of the above hold, then an impasse in problem solving arises. Impasses will be discussed following a short example.

Consider the Missionaries and Cannibals problem. The problem space consists of the different configurations of three missionaries and three cannibals and a boat on the banks of the river. The set of operators is restricted to transporting one or two people at a time because the boat only holds two people. The goal is achieved when all the people have been moved from one side of the river to the other. An additional restriction is that the cannibals can never outnumber the missionaries on one bank of the river because the missionaries will be eaten. In Soar, productions encode all of the necessary knowledge concerning problem selection, operator creation, selection and implementation, state selection and goal achievement. As the first step toward solving this problem in Soar, a production creates an acceptable preference for the Missionaries and Cannibals problem space. Following its selection by the decision procedure, a production fires that creates the initial state. At this point, all relevant operator instances for the current state are created. Assume for the moment that additional productions exist to create preferences so only one operator is clearly best. The decision procedure selects that operator and relevant productions fire, creating a new state with an acceptable preference. This state is selected and the operator slot is cleared because its current value is no longer relevant.

In the above example, we assumed there was sufficient knowledge to select a single operator for each state. When the preferences for a slot do not lead to a clear choice, an *impasse* in problem solving arises and a subgoal is automatically created. An impasse may arise from a tie (the preferences do not determine a best choice), a conflict (the preferences conflict), a rejection (all candidates are rejected), or a no-change (no changes are suggested for any slots). The purpose of the subgoal is to resolve the impasse, possibly by searching for information that will add new preferences and allow a decision to be made. In the subgoal, search in a problem space is used as in the original goal, allowing the full problem-solving capabilities of Soar to be used. Default problem spaces (encoded as productions) are available to handle impasses whenever domain-specific problem spaces are unavailable. A subgoal terminates automatically when new preferences resolve the impasse or lead to a new decision in a higher goal.

Chunking learns by building productions that summarize the processing in a subgoal. The action of a new production is based on a result of a subgoal, while the conditions are based on the pre-impasse working-memory elements that were tested by productions on the path to the generation of the result. Since only the working-memory elements relevant to the results are included, many features of the situation are ignored. When a situation similar to the one that gave rise to the impasse is re-encountered, the chunk will fire, producing the result directly, completely avoiding the impasse and its subgoal.

3 Recovery from Incorrect Knowledge

In this section we describe how recovery from incorrect knowledge is possible in Soar without architectural modification. This will show that the necessary "hooks" already exist in Soar for recovery to take place and that chunking is sufficient to correct errors in long-term knowledge. This section starts with a description of incorrect knowledge in Soar and then proceeds through the five phases of correction: detecting an incorrect decision; forcing reconsideration of the decision; reconsidering the decision; determining the correction; and saving the correction with chunking.

As part of this presentation, a general, domain-independent framework for correcting invalid *control* knowledge is demonstrated. This approach has been implemented within Soar and it will be demonstrated on the Missionaries and Cannibals problem.

3.1 Incorrect Knowledge

Although productions encode all knowledge in Soar, errors only arise through incorrect decisions, that is, because the wrong problem space, state or operator is selected. This leads to an important observation:

- Incorrect knowledge can be corrected by modifying the decisions in which the knowledge is used instead of modifying the productions that encode the knowledge.

Therefore, in Soar we can shift the emphasis in recovery from correcting long-term memory to correcting performance. If Soar learns productions that correct decisions, it will have recovered from incorrect knowledge.

An incorrect decision is caused by an inappropriate preference. We can classify incorrect knowledge based on the types of incorrect preferences: acceptability, necessity, and desirability. Incorrect acceptable preferences correspond to incorrect task knowledge. That is, either the wrong problem spaces, states or operators are created. For example, if an operator is incorrectly implemented, it will produce an acceptable preference for an inappropriate state. This corresponds to having errors in a domain theory in EBL [Mitchell *et al.*, 1986; Rajamoney and DeJong, 1987]. Incorrect necessity preferences correspond to incorrect goal knowledge. Some aspect of the goal is incorrectly encoded so that objects are either required or prohibited inappropriately.

Incorrect desirability and reject preferences correspond to incorrect control knowledge. A simple example of incorrect control knowledge comes from the Missionaries and Cannibals problem. An obvious bit of means-ends control knowledge is to prefer operators that maximize progress toward the goal and minimize movement away from the goal. In states of Missionaries and Cannibal where the boat is on the original side of the river, this leads to a preference for operators that move two people, while if the boat is on the desired side, then preference would be for operators that move one person from the desired side back to the original. Although usually helpful, this knowledge is incorrect in the middle of the problem when two missionaries, two cannibals and the boat are on the desired bank. The means-ends knowledge would prefer sending either one missionary or one cannibal back across the river. In either case, the resulting state would violate the rule that cannibals can not outnumber missionaries. The correct move consists of moving one missionary and one cannibal together across the river.

To recover from this incorrect knowledge, Soar should learn productions that can overcome the preference for moving only one person. This is different than merely learning from failure [Gupta, 1987; Mostow and Bhatnagar, 1987] which in Soar involves learning productions that avoid operators leading to illegal states. If incorrect knowledge is not present, Soar will learn such productions from look-ahead searches [Laird *et al.*, 1984]. But once the incorrect knowledge is present, Soar assumes it is correct and makes decisions without subgoals. To do otherwise would involve questioning every piece of knowledge and negate the advantages of learning.

3.2 Detecting an Incorrect Decision

In Soar, the first step in recovering from incorrect knowledge is to detect that an incorrect decision has been made. This simplifies the credit assignment problem by allowing Soar to detect only incorrect behavior instead of incorrect knowledge. In the Missionaries and Cannibals example, it is easy to detect an incorrect decision because an illegal state is encountered on the path to solution. In this example, general productions test for the invalid state, backtrack to the prior state and force the reconsideration of that decision.

For other tasks, expectation failures, direct feedback from another agent, exhaustion of resources, or other general features of the task may signal that a decision was incorrect. In all cases, productions must detect that an incorrect decision has been made. If the feedback is specific to a previous decision, the situation in which the error occurred can be recreated and reconsidered. If the feedback is not decision-specific, all decisions within a goal can be reconsidered. Decisions likely to be correct can be avoided using domain-specific knowledge or techniques such as dependency-directed backtracking. General mechanisms for determining which of several decisions is in error have not been implemented except for chronological backtracking and the reconsideration of all decisions in a goal. Soar's ability to exhibit a wide variety of methods suggests that using additional techniques should not be problematic[Laird, 1984; Laird and Newell, 1983].

3.3 Forcing Reconsideration

Once an incorrect decision is detected, it is necessary to reconsider the decision and possibly correct it. A decision can be reconsidered by forcing an impasse so that a subgoal arises in which the decision can be made explicitly. Impasses can be forced by adding preferences that cause conflicts. If an acceptable preference or any desirability preferences are suspect, an impasse can be forced by creating a new dummy object with conflicting preferences between it and the suspected objects. If necessity preferences are suspect, additional necessity preferences will force an impasse. In the Missionaries and Cannibals example, an impasse is forced by creating conflicting preferences between the available operators and a dummy operator named *deliberate-impasse*.

3.4 Reconsidering the Incorrect Decision

Once the forced impasse arises, its subgoal provides a context for reconsidering the possibly incorrect decision. It is at this point that other sources of knowledge can be accessed to verify the decision or determine an alternative selection. Other sources of knowledge could be experimentation in the external world [Rajamoney and DeJong, 1987], feedback from another agent, or an appeal to an underlying domain theory [Doyle, 1986] encoded as a problem space. If any of these sources of knowledge is suspect, the same approach can be applied recursively to correct it.

We have not implemented a general approach for obtaining the correct knowledge for task or goal errors. However, for incorrect control decisions, we have implemented a domain-independent approach for correcting decisions based on look-ahead search. This approach is built upon the *selection* problem space, a domain independent problem space that is used as a default whenever impasses arise because of inadequate control knowledge.

The unextended selection space contains operators, called *evaluate-object*, which evaluate the tied or conflicting alternatives. If an evaluation is available via a production, it will be used. If no evaluation is directly available, an impasse arises and a look-ahead search is performed in the resulting subgoal to obtain an evaluation. The resulting evaluations are compared and appropriate preferences are created, thus resolving the impasse.

In those cases where incorrect preferences do exist, the correct choice may not be considered because it is either rejected or dominated by the other objects. To gather information about the rejected and dominated objects, two new operators are added to the selection space: *evaluate-alternatives* and *evaluate-reject*. Evaluate-alternatives, evaluates the alternatives that are not being considered because of possibly incorrect desirability preferences, while

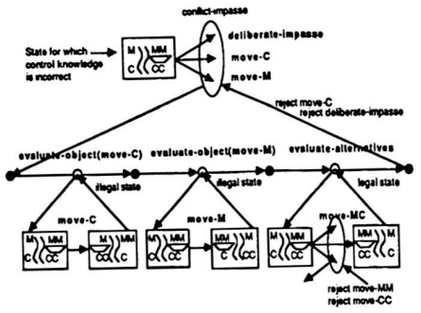

Figure 1: Recovery from incorrect knowledge in Missionaries and Cannibals.

evaluate-reject evaluates the objects that have rejection preferences.

Figure 1 shows a simplified trace of recovery in the Missionaries and Cannibals problem. In this example, the goal is to move all people to the right bank. The current state has one missionary and one cannibal on the left bank, while the boat and the remaining people are on the right bank. The overgeneral knowledge incorrectly prefers moving either one missionary (move-M) or one cannibal (move-C). An impasse is forced using deliberate-impasse. Once in the subgoal, evaluate-alternatives is selected. In the resulting look-ahead search, move-C and move-M are prohibited from being selected so that the best alternative to them will be selected. Although this eliminates two of the operators for the search, the three two-person operators are all available and a tie impasse arises. A look-ahead search is performed for these in a lower selection space (not shown) and the the winner is move-MC. Chunks learned for this selection apply immediately to the top state, so that move-MC is preferred over the other two person operators. Following evaluate-alternatives, moving one cannibal (move-C) as well as moving one missionary (move-M) are evaluated using evaluate-object. These both return a failure evaluation because they generate illegal states in the subgoal. These evaluations are compared to those created for evaluate-alternatives and both move-M and move-C are rejected so that their dominance over the other operators is eliminated. Finally, when all of the evaluation operators are finished, deliberate-impasse is rejected and the impasse is resolved with move-MC being correctly selected.

3.5 Correcting the Decision

If the object preferred by the preferences for a decision is found to be correct, no correction is made. However, if the preferred object is incorrect, then the decision must be corrected. In Soar, a preference can be corrected, either at the decision where the error occurs, or at a decision in a higher goal, (unless the error occurs at the top goal). By changing a higher decision, the preferences for the current decision become irrelevant. This second case is related to the first, because if a decision in a higher goal is changed, then it is as if its prior decision was actually incorrect. However, by correcting a decision in a higher goal, a lower level decision can be avoided that is itself uncorrectable. In either case, the only way to correct a decision is by creating new preferences. Luckily this is sufficient.

The correction of decisions can be divided into three cases based on the type of the incorrect preference. In the first case, an incorrect desirability or acceptable preference leads to an incorrect selection. It may be that an incorrect object is made best, better than, or indifferent to the correct object. Or it may be that the correct object is made worst or worse. In our example from the Missionaries and Cannibals, the incorrect operator is better than the correct operator. All these cases can be corrected by rejecting the incorrect choices. This is done in the selection space when the evaluation created for the not-tied objects (by evaluate-alternatives) is better than the evaluation created for an object about to be selected (by evaluate-object). In this case, a reject preference is created for the incorrectly preferred object. In our example, it is this process that leads to the rejection of the incorrect operators. Once the wrongly preferred object is rejected, other preferences will lead to the selection of the correct object. (These preferences will have been learned in the evaluate-alternatives subgoal.)

In the second case, the correct object is incorrectly rejected or prohibited. This case is important enough to consider with an example. Let's modify our example so that instead of creating better preferences for moving one person back across the river, all operators that move two people are rejected. On the surface this appears to have the same effect, but it makes recovery more difficult. How can we select an operator that has already been rejected? This situation is detected in the selection space when the evaluation produced by evaluate-rejected is better than the evaluation produced for the operator that would have been selected if an impasse had not been forced. The appropriate response, encoded in productions, is to create a new operator with an indirect pointer to the rejected operator. This new operator is made better than the incorrect operator and therefore it is select. This new operator can not be an exact copy of the rejected operator, otherwise the production that rejected the original would also reject it.

The obvious problem is that there must be a general way to apply these new operators, whose only structure is an indirect pointer to another operator. The solution is to select the new operator, fall into a subgoal when no productions fire to apply it, and in the subgoal apply the original, mistakenly reject operator. The original operator can be applied by forcing its selection using a require preference which overrides the rejection. Following its selection, the productions that implement the original operator apply and create a new state that becomes the result of the new operator. Chunking captures this processing, and future applications of the new operator are performed directly by the chunk.

In the third case, an incorrect object is inappropriately required. Neither reject nor prohibit override a require preference because it is encodes knowledge about the validity of the path toward the goal. The only general correction is to modify a higher decision. For example, if an inappropriate operator is required for a state, a new state can be created, using the indirect pointer method described above so that the offending production no longer applies. Modifying a higher decision can also correct the errors described earlier. Flynn and Newell have implemented a scheme where incorrect operators are avoided by creating a new problem space, displacing the old and then using those elements of the old problem space that were still valid [Newell, 1987]. This approach demonstrates the variety of approaches that are possible for recovery from incorrect knowledge.

In summary, incorrect desirability and acceptable preferences are corrected by rejecting the incorrect alternative. Incorrect reject and prohibit preferences are corrected by creating acceptable preferences for new objects that have indirect pointers to the rejected objects. Since all the preferences used in recovery are themselves correctable, any correction can itself be corrected.

3.6 Saving the correction as a permanent repair

Up to this point, we have described how Soar can correct decisions using its problem solving, but we have ignored the process by which a correction can be saved in long-term memory for later use. The solution is simple: once a decision is corrected through the creation of new preferences in a subgoal, chunking learns a new production that will fire under similar circumstances in the future, leading to the correct choice. Most of these productions will fire in parallel with the existing productions. However, when recovery includes creating new objects, the new productions must test for the existence of the rejected objects that they point to. This possibly extends the elaboration phase by one production firing. Interestingly, repeated corrections need not extend it further because all productions that create rejections will fire in parallel, and a second correction need only test for the existence of the original rejected object.

One possible issue is whether the new chunk will be applicable even when an error has not been detected and no attempt is being made to force an impasse. In the implementation of recovery from incorrect control knowledge, the chunks are sufficiently general because the underlying problem solving is independent of the detection of error. All operators in the selection space, such as evaluate-alternatives and evaluate-rejects, are created for every tie or conflict impasse. They are only selected when the evaluations created by evaluate-object operators are insufficient to resolve the impasse.

The other result of the impasse is the rejecting of deliberate-impasse. However, the creation of the reject preference is based on a test of exhaustion, that is, that no more operators are available in the selection space. Tests such as these inherently lead to overgeneralization during chunking [Laird *et al.*, 1986b] so Soar does not create chunks for these results.

Additional chunks are learned for the evaluations computed in the subgoal. These chunks will be available in the future so that the evaluations are computed directly without problem solving. One potential weakness in this approach is that the chunk learned for evaluating alternatives may be overgeneral in that it could apply even if additional alternatives are available. Overgenerality arises because there is an implicit test for exhaustion: the best of all the alternatives was evaluated. Either no chunk should be built or there should be a test that there are no other alternatives available.

4 Results

This section reports the results of using recovery from incorrect knowledge. In Missionaries and Cannibals, without the incorrect knowledge or chunking, the problem is solved in between 178 and 198 decisions, depending on the search. Following learning, the minimum is 22 decisions—a straight line path. Soar learns control knowledge to avoid operators that produce illegal states, as well as learning to select operators that are on the path to solution.

Adding the incorrect control knowledge, but not the recovery knowledge, decreases the number of operators considered at each state, thereby decreasing the search to between 124 and 149 decisions, but an illegal state is always selected at the top-level. After learning, the minimum number of decisions is 25 because the illegal state is created, selected and then rejected. By introducing the recovery knowledge, search before learning is between 159 and 179 decisions. This is less than without the incorrect knowledge, but more than without the recovery code. The important point is that following learning, the minimum is once again 22 decisions. Another test of recovery is to use it after learning has been applied to the incorrect control knowledge. In this case, the problem solving goes up from 25 to 54 decisions, however, after learning the problem solving goes back down to 22. Such a reduction was not possible without recovery.

The productions for controlling recovery from incorrect control knowledge are completely task-independent and they have been used for other simple tasks such as the Eight Puzzle and multi-column subtraction. In the Eight Puzzle, we have added a production that incorrectly prefers to move tiles out of their desired position. This makes it impossible to solve the problem without recovery. In multi-column subtraction, the system incorrectly skips columns as a result of an overgeneral chunk. If it receives negative feedback when it has skipped a column, it backs up and correctly finish the problem.

Table 1 is a summary of example results for these tasks as well as Missionaries and Cannibals. For the Eight Puzzle, it solved a relatively simple problem that requires 12 decisions. For subtraction, the system solves 44-33. The column following the task name contains the number of decisions required to solve the problem if incorrect knowledge is included but the recovery knowledge is not. The next column shows the number of decisions required when the recovery knowledge is added. This usually increases the total time required to solve the problem, however it leads to improved performance after recovery as shown in the final column.

5 Conclusion

We have demonstrated an alternative approach to recovery from incorrect knowledge that does not require deletions or corrections to long-term memory. Instead of modifying long-term memory, the corrections are made during the decisions that arise during the normal course of problem

Task	Without Recovery	With Recovery	After Recovery
Missionaries	25	54	22
Subtraction	13	21	10
Eight Puzzle	∞	45	12

Table 1: Results of using recovery knowledge on three tasks.

solving. The corrections are first determined by problem solving, and then saved as productions by chunking so that the corrections will be available in the future, even before an error in behavior is detected. In addition to demonstrating the feasibility of this approach, we have presented a domain-independent implementation that corrects errors in control knowledge. Within this framework, future research should concentrate on expanding the class of situations in which incorrect decisions can be detected and expanding the sources of knowledge used to verify a decision.

Acknowledgments

Many of the ideas in the paper originated in discussions with Allen Newell, Rex Flynn, Paul Rosenbloom and Olin Shivers. Thanks to Pat Langley and Mark Wiesmeyer for comments on an earlier draft of this paper, and Rob McCarl for his Soar implementations of multi-column subtraction.

References

[Anderson, 1983] J. R. Anderson. *The Architecture of Cognition*. Harvard University Press, Cambridge, MA, 1983.

[Chien, 1987] S. A. Chien. Extending explanation-based learning: Failure-driven schema refinement. In *Proceedings of Third IEEE Conference on AI Applications*, 1987.

[DeJong and Mooney, 1986] G. DeJong and R. Mooney. Explanation-based learning: An alternative view. *Machine Learning*, 1(2):145–176, 1986.

[Doyle, 1986] R. Doyle. Constructing and refining causal explanations from an inconsistent domain theory. In *Proceedings of AAAI-86*. Morgan Kaufmann, 1986.

[Forgy, 1981] C. L. Forgy. Ops5 user's manual. Technical report, Computer Science Department, Carnegie-Mellon University, July 1981.

[Gupta, 1987] A. Gupta. Explanation-based failure recovery. In *Proceedings of AAAI-87*, pages 606–610, Seattle, 1987.

[Hammond, 1986] K. Hammond. Learning to anticipate and avoid planning failures through the explanation of failures. In *Proceedings of AAAI-86*. American Association for Artificial Intelligence, 1986.

[Holland, 1986] J. H. Holland. Escaping brittleness: The possibilities of general-purpose learning algorithms applied to parallel rule-based systems. In *Machine Learning: An Artificial Intelligence Approach, Volume II*, pages 593–624. Morgan Kaufmann Publishers, Inc., Los Altos, CA, 1986.

[Laird and Newell, 1983] J. E. Laird and A. Newell. A universal weak method: Summary of results. In *Proceedings of IJCAI-83*, Los Altos, CA, 1983. Kaufman.

[Laird *et al.*, 1984] J. E. Laird, P. S. Rosenbloom, and A. Newell. Towards chunking as a general learning mechanism. In *Proceedings of AAAI-84*. American Association for Artificial Intelligence, 1984.

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

[Laird *et al.*, 1986b] J. E. Laird, P. S. Rosenbloom, and A. Newell. Overgeneralization during knowledge compilation in Soar. In *Proceedings of the Workshop on Knowledge Compilation*, Otter Crest, OR, 1986. Oregon State University, Oregon State University.

[Laird *et al.*, 1987] J. E. Laird, A. Newell, and P. S. Rosenbloom. Soar: An architecture for general intelligence. *Artificial Intelligence*, 33(3), 1987.

[Laird, 1984] J. E. Laird. *Universal Subgoaling*. PhD thesis, Carnegie-Mellon University, 1984.

[Langley, 1983] P. Langley. Learning effective search heuristics. In *Proceedings of IJCAI-83*, 1983.

[Mitchell *et al.*, 1986] T. M. Mitchell, R. M. Keller, and S. T. Kedar-Cabelli. Explanation-based generalization: A unifying view. *Machine Learning*, 1, 1986.

[Mostow and Bhatnagar, 1987] J. Mostow and N. Bhatnagar. Failsafe — a floor planner that uses EBG to learn from its failures. In *Proceedings of IJCAI-87*, Milano, Italy, 1987. IJCAI.

[Newell, 1987] A. Newell. Unified theories of cognition: 1987 William James lectures. Available on videocassette from Harvard Psychology Department, 1987.

[Rajamoney and DeJong, 1987] S. Rajamoney and G. DeJong. The classification, detection and handling of imperfect theory problems. In *Proceedings of IJCAI-87*, pages 205–207, Milano, Italy, 1987. Morgan Kaufmann.

[Rosenbloom and Laird, 1986] P. S. Rosenbloom and J. E. Laird. Mapping explanation-based generalization onto Soar. In *Proceedings of AAAI-86*, Philadelphia, PA, 1986. American Association for Artificial Intelligence.

[Rosenbloom *et al.*, 1987] P. S. Rosenbloom, J. E. Laird, and A. Newell. Knowledge-level learning in Soar. In *Proceedings of AAAI-87*. American Association for Artificial Intelligence, American Association for Artificial Intelligence, 1987.

[Rosenbloom *et al.*, 1988] P. S. Rosenbloom, J. E. Laird, and A. Newell. The chunking of skill and knowledge. In *Working Models of Human Perception*. Academic Press, London, 1988. In press.

[Steier *et al.*, 1987] D. Steier, J. E. Laird, A. Newell, P. S. Rosenbloom, et al. Varieties of learning in Soar: 1987. In P. Langley, editor, *Proceedings of the Fourth International Workshop on Machine Learning*. Kluwer, 1987.

Comparison of the Rete and Treat Production Matchers for Soar (A Summary)

P. Nayak, Stanford University, A. Gupta, Stanford University, and P. S. Rosenbloom, USC–ISI

Abstract

RETE and TREAT are two well known algorithms used for performing match in production systems (rule-based systems). In this paper, we compare the performance of these two algorithms in the context of Soar programs. Using the number of tokens processed by each algorithm as the performance metric, we show that the RETE algorithm performs better than the TREAT algorithm in most cases. Our results are different than the ones shown by Miranker for OPS5. The main reasons for this difference are related to the following: (i) fraction of times no joins need to be done; (ii) the long chain effect; (iii) matching of static structures; and (iv) handling of combinatorial joins. These reasons go beyond Soar in their applicability, and are relevant to other OPS5-based production systems that share some of Soar's properties. We also discuss several implementation issues for the two algorithms.

1 Introduction

Soar is a cognitive architecture that provides the foundations for building systems that exhibit general intelligent behavior [Laird *et al.*, 1987]. Soar uses an OPS5-like production system [Brownston *et al.*, 1985] to encode its knowledge base and it provides a vision of how future expert systems may be constructed. It has been exercised on many different tasks, including some of the classic AI toy tasks such as the Eight Puzzle, and the Missionaries and Cannibals problem, as well as on large tasks such as the R1 computer configuration task [Rosenbloom *et al.*, 1985], the Neomycin medical diagnosis task [Washington and Rosenbloom], and the Cypress algorithm design task [Steier, 1987]. It exhibits a wide range of problem-solving mechanisms and has a general mechanism for learning.

RETE and TREAT are prominent algorithms that have been designed to perform match in production systems. The RETE algorithm [Forgy, 1982] was proposed by Forgy and is currently used in almost all implementations of OPS5-like production systems. The TREAT algorithm [Miranker, 1984] has been proposed more recently by Miranker. In his recent study [Miranker, 1987], Miranker presents empirical evidence, based on five OPS5 programs, that seem to show that the TREAT match algorithm can outperform the RETE match algorithm, often by more than fifty percent.

This paper describes our experiments with the TREAT match algorithm for Soar, and the results of comparing its performance with the currently used RETE algorithm [Scales, 1986]. Our experiments show very different results than Miranker's. They show that for Soar, RETE outperforms TREAT in most cases. The main reasons for this difference are related to the following: (i) fraction of times no joins need to be done; (ii) the long chain effect; (iii) matching of static structures; and (iv) handling of combinatorial joins. These reasons go beyond Soar in their applicability, and are relevant to other OPS5-based production systems that share some of Soar's properties. We also discuss several implementation issues for the two algorithms.

The paper is organized as follows. The next section presents background material on Soar, RETE, and TREAT. Section 3 describes the condition ordering algorithms used in the various match algorithms. Section 4 presents the results and the discussion. The conclusions are in section 5.

2 Background

2.1 Soar

Soar is an architecture for a system that is to be capable of general intelligence. Soar is based on the *Problem Space Hypothesis*, which states that all symbolic goal-oriented behavior can be cast as a search in a problem space. A problem space consists of a set of states and a set of operators to move amongst these states. Every goal in Soar is formulated as a search in some appropriate problem space. Goals may have subgoals, leading to a hierarchy of goals.

Long-term knowledge in Soar is stored in OPS5-like productions. The current status of the problem-solving is stored in the *working memory*. The working memory consists of the *context stack*, the *augmentations*, and the *preferences*. The context stack contains the hierarchy of goals, with each goal having slots for the associated problem-space, state, and operator. Problem solving is driven by selecting objects for the slots in the context. Augmentations specify values for an attribute of an object. Preferences encode statements about selection of objects for the slots in the context stack.

To better understand the match algorithms, we now describe the working memory elements and the productions in some detail [Laird, 1986]. Augmentations of objects have four fields: (i) the *class* of the object, (ii) its (unique) *identifier*, (iii) the name of some *attribute*, and (iv) a *value* for that attribute. Preferences have nine fields, as against only four for augmentations. The meanings of these fields are not very relevant to the paper and will not be discussed here.

A production in Soar is a condition-action rule. The condition part of a production is made up of *condition elements*. Each field of a condition element specifies tests on the corresponding fields of a working memory element. A working memory element *matches* a condition element if it satisfies all

*This research was sponsored by the Hughes Aircraft Company, by Digital Equipment Corporation, and by the Defense Advanced Research Projects Agency (DOD) under contracts N00039-86-C-0133 and MDA903-83-C-0335. 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 Hughes Aircraft Company, Digital Equipment Corporation, the Defense Advanced Research Projects Agency, the US Government, or the National Institutes of Health.