ieved in some Soar tasks. A possible avenue of investigation is to equip the system with diagnostic tools to automatically deduce the causes of the low speedups. For example, to identify long chains, the system can look at the last few node activations on the cycles with low parallelism. The system can then make adaptive changes, such as introducing bilinear networks, to increase the speedups. Other areas for future study include the effects of chunking over long periods of time on parallelism. A longer term goal includes the optimization of the Lisp-based portion of the system, its conversion to C and parallelizing other areas of the system besides match.

8. Summary

In this paper, we have explored techniques for efficient parallel implementation of Soar, a significant AI system. This provided a unique opportunity to study the match parallelism of a learning production system. We presented techniques for adding productions and updating their state at run-time. We presented the speedups obtained in the match on our system and the effects of chunking on the speedups. We showed that Soar/PSM-E is capable of achieving significant speedups. The discussion of the impact of chunking on parallelism indicates that the opportunities for exploiting parallelism should increase a great deal in Soar systems that add a large number of chunks. We also analyzed speedups in detail and showed that two effects limit the parallelism in the system: short cycles and long chains. Some solutions to these problems were proposed, which would increase the speedups achievable. However, other modules in Soar still need to be optimized for this system to be useful as a real engine for Soar users.

Acknowledgement

We thank John Laird, Paul Rosenbloom and Peter Steenkiste for helpful comments on earlier drafts of this paper. We also thank Kathy Swedlow for her technical editing.

This research was sponsored by Encore Computer Corporation, Digital Equipment Corporation and by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 4976 under contract F33615-87-C-1499 and monitored by the:

Avionics Laboratory
Air Force Wright Aeronautical Laboratories
Aeronautical Systems Division (AFSC)
Wright-Patterson AFB, OHIO 45433-6543

Anoop Gupta is supported by DARPA contract MDA903-83-C-0335 and an award from the Digital Equipment Corporation. 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 Encore Computer Corporation, Digital Equipment Corporation and the Defense Advanced Research Projects Agency or the US Government.

References

1. Fikes, R., Hart, P., and Nilsson, N. "Learning and Executing Generalized Robot Plans". *Artificial Intelligence 3*, 1 (1972), 251-288.

2. Forgy, C. L. OPS5 User's Manual. Tech. Rept. CMU-CS-81-135, Computer Science Department, Carnegie Mellon University, July, 1981.

3. Forgy, C. L. "Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem". *Artificial Intelligence 19*, 1 (1982), 17-37.

4. Forgy, C. L. The OPS83 Report. Tech. Rept. CMU-CS-84-133, Computer Science Department, Carnegie Mellon University, May, 1984.

5. Gupta, A.. *Parallelism in Production Systems.* Morgan-Kaufman, Los Altos, California, 1987.

6. Gupta, A., Forgy, C. L., Kalp, D., Newell, A., & Tambe. M. Results of Parallel Implementations of OPS5 on the Encore Multiprocessor. Proceedings of the International Conference on Parallel Processing, August, 1988. To appear.

7. Laird, J. E. Universal Subgoaling. In Laird, J.E., Rosenbloom, P.S., and Newell, A., Ed., *Universal Subgoaling and Chunking: The Automatic Generation and Learning of Goal Hierarchies*, Kluwer Academic Publishers, Boston, Massachusetts, 1986, pp. 1-131.

8. Laird, J. E., Newell, A., and Rosenbloom, P. S. "Soar: An Architecture for General Intelligence". *Artifical Intelligence 33*, 1 (1987), 1-64.

9. Laird, J. E., Rosenbloom, P. S., & Newell, A. "Chunking in Soar: The Anatomy of a General Learning Mechanism". *Machine Learning 1*, 1 (1986), 11-46.

10. Lehr, T. F. The Implementation of a Production System Machine. Proceedings of the Hawaii International Conference on Systems Sciences, January, 1986, pp. 177-205.

11. Minton, S. Selectively Generalizing Plans for Problem-solving. Proceedings of the Ninth International Joint Conference on Artificial Intelligence, August, 1985, pp. 596-599.

12. Miranker, D. P. Treat: A Better Match Algorithm for AI Production Systems. Proceedings of AAAI-87, August, 1987, pp. 42-47.

13. Newell, A. Reasoning, Problem Solving and Decision Processes: The Problem Space as a Fundamental Category. In Nickerson, N., Ed., *Attention and Performance VIII*, Lawrence Erlbaum and Associates, Hillsdale, New Jersey, 1981, pp. 693-718.

14. Newell, A. Unified Theories of Cognition. The William James Lectures. Harvard University. Available in videocassette from Harvard Psychology Department.

15. Oflazer, K. Partitioning in Parallel Processing of Production Systems. Tech. Rept. CMU-CS-87-114, Computer Science Department, Carnegie Mellon University, March, 1987.

16. 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.

17. Schreiner, F. , Zimmerman, G. Pesa-1- A Parallel Architecture for Production Systems. Proceedings of the International Conference on Parallel Processing, August, 1987, pp. 166-169.

18. Steier, D. M. CYPRESS-Soar: A Case Study in Search and Learning in Algorithm Design. Proceedings of the Tenth International Joint Conference on Artificial Intelligence, August, 1987, pp. 327-330.

19. Steier, D. E., Laird, J. E., Newell, A., Rosenbloom, P. S., Flynn, R. A., Golding, A., Polk, T. A., Shivers, O. G., Unruh, A., & Yost, G. R. Varieties of Learning in Soar: 1987. Proceedings of the Fourth International Workshop on Machine Learning, June, 1987, pp. 300-311.

20. Tambe, M. & Newell, A. Why Some Chunks Are Expensive. Proceedings of the Fifth International Workshop on Machine Learning, June, 1988. To appear.

21. Tenorio, M. F. M. and Moldovan, D. E. Mapping Production Systems Into Multi-processors. Proceedings of the International Conference on Parallel Processing, August, 1985, pp. 56-62.

CHAPTER 29

Applying Problem Solving and Learning to Diagnosis[1]

R. Washington, Stanford University, and P. S. Rosenbloom, USC–ISI

Abstract

Classification and construction tasks have been considered qualitatively different problems for AI systems. This paper shows that the two problems can be built out of the same primitive actions. Furthermore, it describes Neomycin-Soar, a system for classification implemented in Soar, a framework that has been used to study construction tasks. In addition, the effects of Soar's learning capability on classification tasks are explored.

1 Introduction

Classification and construction are two classes of problems that AI systems have been built to solve. The classes have been considered to be qualitatively different, with each having particular solution methods [Clancey, 1985]. However, classification and construction can each be viewed as the composition of the same fundamental problem-solving mechanisms. This paper will describe a system for classification, specifically medical diagnosis, that has been built in a framework that has been used for construction problems.

[1]This research was sponsored by the Defense Advanced Research Projects Agency (DOD) under contract number N00039-86C-0033 and by an NSF fellowship. 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, or the US Government.

Construction problem solving entails designing a solution from primitive elements. Examples of construction problems are computer design [McDermott, 1982] and automatic programming [Kant, 1985]. Construction can be viewed as a process that iterates over some basic activities. A set of operators, appropriate for the given situation, is generated. The best of these operators is selected. Finally, the chosen operator is executed, leading to a new situation. This generate-select-execute loop continues until the solution is constructed. In addition to the work going on in this basic processing loop, the operators may themselves be complex, and need to be implemented by recursive use of the generate-select-execute loop with a set of suboperators. Likewise, the selection process may be complex, requiring for example a lookahead search to determine which operator to select (which is itself pursued via the generate-select-execute loop).

Classification problem solving, on the other hand, involves choosing a solution from a pre-enumerated set of alternatives. The choice of solution is based on a set of data about the situation, some of which may be available at the beginning of problem solving, and the rest of which must be determined as needed. This method has been applied in tasks ranging from structural analysis [Bennett *et al.*, 1978] to circuit diagnosis [Brown *et al.*, 1982] to medical diagnosis [Buchanan and Shortliffe, 1984, Pople, 1977]. But classification can be viewed in terms of the same primitive activities as construction. Potential solutions for the problem are generated. The best of these is then selected. (Execution is not normally involved unless something is actually done with the solution.) Each of these steps might require extensive problem solving. Potential solutions may be generated based on the initial data (for instance, by associational rules). More data might become necessary to successfully generate the potential solutions — for instance, an association between known data and a potential solution may depend on something currently unknown — in which case the set of known data will grow. These new data may change the set of potential solutions. Once a set of competing solutions is available, the problem is to select among them. This may require yet more data. As before, any change in the set of known data may generate different competing solutions. This process of generation and selection continues until the competing set of solutions is reduced to an acceptable size, usually either a single solution or a minimal set of solutions that explain the known data.

A problem-solving framework can support both construction and classification systems if it supports generation, selection, and execution as basic activities. In service of these activities it needs to allow an arbitrary amount of processing that consists of further generation, selection, and execution. One such framework is the Soar problem-solving architecture [Laird *et al.*, 1987].

Soar is based on formulating all symbolic goal-oriented behavior as search in problem spaces. The primitive acts of the system, called *decisions*, are those required to pursue this search: the generation and selection of problem spaces, states, and operators, plus the application of operators to states to create new states. Object generation, object selection, and operator application correspond to the basic activities of generation, selection, and execution that comprise construction and classification. The information necessary for the performance of these basic activities is generated in one of two ways: by the firing of productions, or by the recursive use of problem space search in subgoals. Subgoal search results when the knowledge provided by production-firing is inadequate to make a decision. This leads to hierarchical processing, where each level of processing can involve arbitrary amounts of generation, selection, and execution. The subgoal search terminates when results are returned that allow a decision to be made. Learning occurs by converting subgoal-based search into rules that generate comparable results under similar conditions. Specifically, for each result returned, a production is built by first capturing the conditions that the result depends upon, and then generalizing these conditions. This *chunking* process is a form of explanation-based learning in which the explanation is derived from a trace of the search that led to the results of the subgoal [Rosenbloom and Laird, 1986].

Soar has been used as a framework for building construction systems. VAX computer configuration is the domain of R1-Soar [Rosenbloom *et al.*, 1985], based on the R1 expert system. Automatic programming has been addressed by Cypress-Soar [Steier, 1987] and Designer-Soar [Steier and Newell, 1988]. Since the integration of problem solving and learning is a major theme of Soar, these systems had the opportunity to investigate the effects of learning in their tasks.

The system described here is a classification system built in Soar. The system, Neomycin-Soar, is based on Neomycin [Clancey and Letsinger, 1981, Clancey, 1984], an expert system for diagnosis of infectious diseases. Neomycin arose from the realization that the lack of explicit control over the reasoning performed in Mycin — where domain and control knowledge were competely conflated — was a major problem when the system was used for explanation and tutoring [Buchanan and Shortliffe, 1984]. Neomycin was designed to separate out the control knowledge from the domain knowledge. An explicit diagnostic strategy, consisting of a set of hierarchically-structured procedures, controls the use of the domain knowledge. Neomycin attempts to generate possible diagnoses by forward reasoning from the initial data. When it needs additional data to support these implications, the system determines the needed data by backward chaining and question asking. Once a set of

hypotheses has been generated, Neomycin tries to discriminate among them, increasing or decreasing belief in some of these hypotheses by further backward chaining and question asking. Neomycin-Soar re-implements a portion of Neomycin sufficient to address the issues of classification problem solving.

The basic issues addressed in Neomycin-Soar are the ability of a framework like Soar to support classification problem solving, and the effects of learning on classification tasks. The description of the system is divided into two major components: the representation of the domain knowledge (Section 2); and the implementation of a diagnostic strategy that is based on the model of classification problem solving (Section 3). Section 4 presents the results of learning in this task, examining the kinds of knowledge learned and the effects of transferring this knowledge to similar tasks. Finally, section 5 summarizes the main results of the system.

2 Domain Knowledge Representation

Knowledge in Soar is stored in productions. The actions of a production contain the knowledge, while the conditions specify the situation in which the knowledge can be retrieved. The knowledge in Neomycin-Soar is an adaptation of the declarative knowledge structures in Neomycin to this production model of knowledge storage and retrieval.

Two basic representations comprise the knowledge in Neomycin. Generalization hierarchies record the subsumption relations among the data and are used for belief inheritance. Associational rules capture empirical relations among data. The rules follow the format of Mycin rules, which are condition-action rules with associated certainty factors.

The inheritance hierarchies are represented in Neomycin-Soar by productions that store individual fragments of a hierarchy. When a datum is under consideration, a production will fire, producing information about the datum's parents and children in the hierarchy. When the system considers one of these new data, the system will generate information about the new datum's parents and children. Thus only the relevant portion of the hierarchy is available at any time. There are an average of approximately 5 inheritance productions for any datum: there are on average 2.5 parents and children for a datum, and inheritance is used two ways — the datum both inherits belief from other objects in the hierarchy and propagates belief to the other objects.

Neomycin's domain rules are represented as operators in Neomycin-Soar. Such an operator is proposed by any of a set of productions, reflecting the different situations in which the rule can apply. In addition, another set of

antecedent	Definitional, relates premises to conclusion with absolute certainty
trigger	Strong relationship between premises and actions
soft-data	All of the premises can be easily determined
lab-data	Some of the premises are laboratory data

Figure 1: Classes of Neomycin rules.

productions implements the operator. The certainty factor is stored as a number, and the propagation of belief is processed by a set of productions separate from the ones that process the purely symbolic effects. A typical Neomycin domain rule translates to about ten Neomycin-Soar productions. This is due to the combination of two factors. The rules in Neomycin are categorized into different classes (see Figure 1), and a rule may belong to more than one class. As part of its diagnostic strategy, Neomycin examines some classes of rules before others, so the rule should be proposed only when it belongs to the class being examined. In addition, the rule may have multiple premises. The system may know about one of the premises and attempt to derive the consequences of it using the rule, so the rule should be suggested when any of the premises is being considered. Figure 2 shows the way a Neomycin domain rule is represented in Neomycin-Soar. On average, the Neomycin rules have 1.7 conditions and belong to 1.8 classes of rules.

Neomycin's domain knowledge covers 288 different parameters, related by hierarchical relationships, and 169 associational rules. Representing this knowledge as productions leads to a total of 4548 domain productions in Neomycin-Soar, 2991 associational productions and 1557 inheritance productions. There is a high degree of similarity among the productions, which is exploited in Soar's internal representation of the productions, so the system is able to work with that number of productions. The reason that the average for domain rules in the complete system exceeds ten is that some Neomycin rules stretch the Mycin format, compiling a large number of rules into a single, tabular rule. If the distinct cases within the rule are counted separately, the average appears much closer to ten.

3 Diagnostic Strategy

Neomycin-Soar has a top-level problem space that reflects the basic generation and selection processes in classification. Possible hypotheses are represented as operators in this space. Generation consists of proposing operators, and

Operator-proposal productions		
function	number of productions	explanation
forward	pc	when finding the implications of one of the premises and one of the classes, suggest the rule
backward	1	when finding support for the action, suggest the rule

Operator-implementation productions		
function	number of productions	explanation
satsify-premise	p	notice when premise is satisfied
explore-premise	p	when operator does not apply, set up sub-operator to find out about this premise
implication	1	when all premises are satisfied, add action to known data
certainty factor	2	when rule fires, add certainty factor information

Figure 2: Representation of a Neomycin domain rule in Neomycin-Soar. In this figure, p is the number of premises for the rule, and c is the number of classes the rule belongs to.

selection is deciding among the competing operators. The hierarchical subprocedures in Neomycin that generate hypotheses are mapped into a hierarchical set of problem spaces for hypothesis-generation in Neomycin-Soar — the basic steps of the subprocedures map onto more problem-space operators, and subprocedure control maps onto additional operator selection knowledge (encoded as productions). The selection subspaces are not implemented, but would use many of the same subprocedures.

Neomycin-Soar follows the basic paradigm of generating hypotheses by reasoning forward from the known data. When a rule is important enough, as defined by the class of rules it belongs to, then an unknown premise of the rule will cause backward chaining to determine its truth. In Neomycin-Soar, this "backward chaining on rules" really consists of subgoaling on operators; that is, if an operator is selected with unmet preconditions, a subgoal is generated in which the system attempts to satsify the preconditions. This process continues recursively, generating additional subgoals as necessary, until known data are found or questions are asked. In Neomycin-Soar (as well as Neomycin) this backward chaining can be expensive, as it is exhaustive, so the search order is controlled by bringing additional control knowledge to bear on the selection process. Information is first sought from the inheritance hierarchy. If that fails, then rules are followed backward in order of importance. When no rules contribute any belief to the premise, then a question is posed, asking the user for information. Variations on this are possible: some data are marked so that a question is asked before the system backward chains on rules, and some data may not be asked.

The diagnostic strategy in Neomycin-Soar consists of a fixed set of 263 productions. When combined with the 4548 domain productions, Neomycin-Soar is able to successfully reproduce Neomycin's hypothesis generation behavior.

4 Learning

Since learning is a fundamental part of the Soar architecture, the implementation of a version of Neomycin in Soar gave us the opportunity to examine the effects of learning on the diagnosis process. Chunking is an automatic learning process designed to capture the results of subgoal-based reasoning in new productions. Neomycin-Soar uses subgoals to perform complex generation, selection, and execution activities, so any of this information can potentially be learned.

One of the expectations about learning in Neomycin-Soar was that the system would generate Mycin-like rules, with control and domain knowledge

IF trying to derive consequences of tense-fontanel,
and trying to apply an operator to look for trigger rules,
THEN reject operator (since there are no appropriate trigger rules)

Figure 3: Sample chunk to avoid a dead-end.

IF trying to apply RULE272,
and nothing is known about ctscan-ventricular-size,
and no input exists about ctscan-ventricular-size,
THEN ask a question about ctscan-ventricular-size

Figure 4: Sample chunk to suggest asking a question.

compiled together. However, none of the learned rules resemble Mycin rules. The reason is that Soar compiles into its chunks all the information that subgoal results depend on. In the case of domain rules, the effects of the rules depend not just on the rule and the control, as in Mycin, but also on any information used to support the rule's premises. So the chunk learned for one rule might also contain all the information about other rules fired in support of the original rule. The resulting chunk appears to be similar to a number of Mycin rules mixed together.

Still, some interesting chunks were learned. One of the more common and more useful types of chunks stores the conditions under which following a path of reasoning will lead to no new information (Figure 3). These productions save future work by avoiding dead-ends.

Another class of chunks stores the conditions under which a question should be asked (Figure 4). These chunks can cause relevant questions to be asked earlier in the diagnosis process than they would be normally.

A third class of chunks records the consequences of a particular set of data. Combined with the chunks for asking questions, these chunks can avoid a great deal of work before and after asking a question (Figure 5). In cases where the answer to the question differs from the case the chunks have stored, the chunks will not apply, and the system will do its usual processing to determine the consequences of the answer. If the answers do correspond to cases that the system has learned, the chunks move the system directly from question to question, as in Figure 6.

One difficulty with chunking in Neomycin's domain involves the numerical representation for certainty factors. Since certainty factors are represented as numbers, with no underlying knowledge about their significance, the exact numbers remain in the chunks. The resultant chunks are overspecific, not

IF trying to apply RULE272,
and nothing is known about ctscan-ventricular-size,
and there is input confirming ctscan-ventricular-size,
THEN create a new state

IF trying to apply RULE272,
and nothing is known about ctscan-ventricular-size,
and there is input confirming ctscan-ventricular-size,
and a new state has been created,
THEN add ctscan-ventricular-size to the list of known data
and make its certainty factor 1.0

Figure 5: Sample chunks to generate results of an answer.

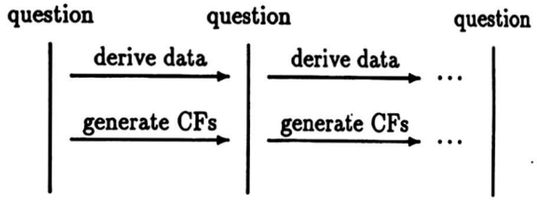

Figure 6: System behavior when chunks apply maximally.

	decisions without transfer	decisions with transfer
case A	625	107
case B	1014	504
case C	966	652
case D	766	475
case E	647	194

Table 1: Comparison of decisions with and without added chunks.

applying in situations where the general result holds but the certainty factor differs. One exception to this is that Neomycin-Soar makes use of the concept of a "significant" certainty factor when deciding whether or not to apply a rule. This gets reflected in the chunks, where some of the chunks check a certainty factor only for significance and not for precise numbers.

To test the utility of the learned knowledge, five cases were chosen from the Neomycin case library. Chunking was used on one case, which will be called case A, and the resultant chunks were applied to a rerun of case A and to the remaining four cases, referred to as case B, case C, case D, and case E.

Since the system with all of the domain knowledge was slow to run — a sample run took 3.75 hours — the domain knowledge base was pared down to a set of data and rules appropriate for these cases. In addition, studying the details of the chunking behavior is more tractable with a smaller set of productions. The learning results gathered so far should apply to the full knowledge base, however, and such a set of tests is planned for the entire knowledge base. The knowledge base for the test reported here consisted of 29 Neomycin rules, producing 395 Neomycin-Soar associational productions, and 40 Neomycin data, producing 205 Neomycin-Soar inheritance productions. The strategy knowledge remained unchanged.

Chunking in case A produced 340 new productions, which were then loaded into Soar when each of the cases was run. As can be seen from Table 1, the number of decisions to reach a solution decreased significantly when the chunks were used, ranging from 33% to 70% speedup for cases B–E[2]. Case A showed an 83% speedup in decisions.

The price paid for the reduced number of decisions is that matching is slower with the extra productions. In fact, the elapsed times of cases B–E after the chunks were added show very little improvement, with changes

[2]These statistics are preliminary. The exact numbers may change while the system is revised, but the general results should remain the same.

	seconds without transfer	seconds with transfer
case A	222.3	92.7
case B	413.9	416.8
case C	343.0	419.9
case D	266.8	300.4
case E	210.2	146.7

Table 2: Comparison of elapsed time with and without added chunks.

ranging from a 22% slowdown to a 30% speedup[3]. Case A showed a significant improvement of 73%, as would be expected with a high amount of transfer. The numbers can be seen in Table 2. We expect the problems with chunking-induced slowdowns to be eliminated when a solution to the problem of expensive chunks — that is, chunks that require a combinatorial match process [Tambe and Newell, 1988] — is implemented, along the lines presented in [Tambe and Rosenbloom, 1988].

Preliminary analysis shows that avoiding dead ends in diagnosis is the major source of speedup in transfer tasks. So it is quite possible that using the full knowledge base, with more knowledge irrelevant to the cases tested, might provide more opportunity for speedups. Further work is planned to determine whether elapsed-time speedup can be achieved, and to characterize the conditions under which learning will prove useful.

5 Conclusion

Classification and construction tasks can be built from the same fundamental activities of generation, selection, and execution. This level of description lies between the level of a general purpose language and a specific shell. By capturing the common elements of diverse problem-solving techniques, it may supply a useful framework for describing expert system applications.

A specific system for classification, Neomycin-Soar, was developed to implement these ideas. The framework in which it was built, Soar, has previously been shown to support construction tasks. Neomycin-Soar shows Soar's applicability to classification tasks.

Furthermore, learning was applied to the system to understand its effect on classification problem solving. Although the initial expectations about what would be learned proved to be incorrect, the system learns useful information

[3]These statistics were gathered from runs on a TI Explorer II with 8 megabytes of real memory.

that transfers to other cases. In particular, the knowledge necessary to avoid dead-ends contributes significantly to speeding up problem solving. Knowledge about when to ask questions and what results the answers imply has the potential to carry the problem solving from question to question, with little or no complex processing in between.

Certainty factors interact with learning, making some of the learned productions overspecific. The lack of underlying knowledge to support the numeric beliefs causes the numbers to be left in productions which otherwise are applicable to a range of cases.

The domain knowledge in Neomycin-Soar is structured as productions. Each production has a small amount of knowledge stored along with specific situations in which it is applicable. This approach led to a fairly large expansion in the number of elements (productions or rules) required to store the data. An alternative approach would have been to store the rules with less specific access conditions and more declarative information in the actions. A general rule interpreter in the system could use the information to apply the rules. Chunking would then create new productions that reflect how the rules are used. The advantage of such an approach is that it starts out with fewer productions and only compiles the rules for the cases it encounters. The disadvantage is that the new compiled productions are likely to be more specific than the hand-encoded ones, so it would take more productions to achieve the same behavior as the current system on all cases.

6 Acknowledgements

We would like to thank Bill Clancey for helping us understand Neomycin and for his interest in this project. Thanks also to the members of the Soar research group for their comments on this work as it developed.

References

[Bennett *et al.*, 1978] J. Bennett, L. Creary, R. Engelmore, and R. Melosh. *SACON: A knowledge-based consultant for structural analysis.* Technical Report HPP 78-23, Stanford University, 1978.

[Brown *et al.*, 1982] J. S. Brown, R. R. Burton, and J. de Kleer. Pedagogical, natural language, and knowledge engineering techniques in sophie i, ii, and iii. In D. Sleeman and J. S. Brown, editors, *Intelligent Tutoring Systems*, pages 227–282, Academic Press, 1982.

[Buchanan and Shortliffe, 1984] B. G. Buchanan and E. H. Shortliffe, editors. *Rule-Based Expert Systems: The MYCIN Experiments of the Stanford Heuristic Programming Project.* Addison-Wesley, Reading, MA, 1984.

[Clancey, 1984] W. J. Clancey. *Acquiring, representing, and evaluating a competence model of diagnosis.* Technical Report HPP84-2, Stanford University, February 1984.

[Clancey, 1985] W. J. Clancey. *Heuristic Classificiation.* Technical Report HPP85-5, Stanford University, March 1985.

[Clancey and Letsinger, 1981] W. J. Clancey and R. Letsinger. Neomycin: Reconfiguring a rule-based expert system for application to teaching. In *Proceedings of IJCAI-81*, pages 829–836, August 1981.

[Kant, 1985] E. Kant. Understanding and automating algorithm design. *IEEE Transactions on Software Engineering*, 11:1361–1374, 1985.

[Laird *et al.*, 1987] J. E. Laird, A. Newell, and P. S. Rosenbloom. Soar: An architecture for general intelligence. *Artificial Intelligence*, 33(1):1–64, September 1987.

[McDermott, 1982] J. McDermott. R1: A rule-based configurer of computer systems. *Artificial Intelligence*, 19:39–88, 1982.

[Pople, 1977] H. E. Pople. The formation of composite hypotheses in diagnostic problem solving: An exercise in synthetic reasoning. In *Proceedings of IJCAI-77*, pages 1030–1037, 1977.

[Rosenbloom and Laird, 1986] P. S. Rosenbloom and J. E. Laird. Mapping explanation-based generalization onto soar. In *Proceedings of AAAI-86*, Philadelphia, 1986.

[Rosenbloom *et al.*, 1985] P. S. Rosenbloom, J. E. Laird, J. McDermott, A. Newell, and E. Orciuch. R1-soar: An experiment in knowledge-intensive programming in a problem-solving architecture. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 7(5):561–569, 1985.

[Steier and Newell, 1988] D. Steier and A. Newell. Integrating multiple sources of knowledge into designer-soar, an automatic algorithm designer. In *Proceedings of the Seventh National Conference on Artificial Intelligence (AAAI-88)*, pages 8–13, 1988.

[Steier, 1987] D. M. Steier. A case study in search and learning in algorithm design. In *Proceedings of IJCAI-87*, pages 327–330, 1987.

[Tambe and Newell, 1988] M. Tambe and A. Newell. Some chunks are expensive. In J. Laird, editor, *Proceedings of the Fifth International Conference on Machine Learning*, pages 451–458, Ann Arbor, 1988.

[Tambe and Rosenbloom, 1988] M. Tambe and P. S. Rosenbloom. *Eliminating Expensive Chunks.* Technical Report #88-189, Carnegie-Mellon University Computer Science Department, 1988.