Learning 10,000 Chunks: What's it Like Out There?

Bob Doorenbos, Milind Tambe, and Allen Newell

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213-3890
Robert.Doorenbos@CS.CMU.EDU, Milind.Tambe@CS.CMU.EDU, and Allen.Newell@CS.CMU.EDU

Abstract

This paper describes an initial exploration into large learning systems, i.e., systems that learn a large number of rules. Given the well-known utility problem in learning systems, efficiency questions are a major concern. But the questions are much broader than just efficiency, e.g., will the effectiveness of the learned rules change with scale? This investigation uses a single problem-solving and learning system, Dispatcher-Soar, to begin to get answers to these questions. Dispatcher-Soar has currently learned 10,112 new productions, on top of an initial system of 1,819 productions, so its total size is 11,931 productions. This represents one of the largest production systems in existence, and by far the largest number of rules ever learned by an AI system. This paper presents a variety of data from our experiments with Dispatcher-Soar and raises important questions for large learning systems.[1]

Introduction

The machine learning community has a strong view that it is infeasible simply to learn new rules, because the cost of matching them would soon devour all the gains. This is known as the *utility problem* (Minton, 1988a) — the cumulative benefits of a rule should exceed its cumulative computational cost. There is data that supports this, as well as a special phenomenon of expensive chunks (Tambe et al., 1990). Thus, when considering systems that are to learn indefinitely, efficiency issues are a critical concern.

But the issues are broader than just efficiency. As we grow large systems via learning, what sort of systems will emerge? Will they be able to keep learning? What will happen to the usefulness of the learned rules, e.g., will only a few rules provide all the action? Will there be mutual interference? What scales and what doesn't? It is not even clear these are the right questions, because we do not know what such systems will be like.

This paper reports on an empirical exploration to begin to get answers to these questions, both the pressing ones of efficiency and the myriad others that should interest us. It reports on a single problem-solving and learning system, Dispatcher-Soar, performing a sequence of randomly generated tasks, all the while continuously learning. At the time of this paper, Dispatcher-Soar has learned 10,112 new rules (called *chunks*), on top of an initial system of 1,819 rules, so its total size is 11,931 rules. This represents one of the largest rule-based systems in existence, and by far the largest number of rules ever learned by an AI system.[2]

This effort is only an initial probe into the unknown of large learning systems. We have no intention of stopping at 10,000 chunks. Will matters look the same at 50,000 chunks? Dispatcher-Soar is only a single system working in a single task domain. What is idiosyncratic to this system and task domain, and what is more general? Indeed, what are the right questions to be asked of these systems and task domains to better understand these issues? Thus, although we present data from a world none of us has yet glimpsed, the paper mostly focuses on formulating the right questions.

Dispatcher-Soar

A brief description of Dispatcher-Soar will suffice. The system was not designed with these experiments in mind, but to explore how to use databases. Its task is to dispatch messages for a large organization, represented in a database containing information about the people in the organization, their properties, and their ability to intercommunicate. Given a specification of the desired set of message recipients (e.g., "everyone involved in marketing for project 2"), the system must find a way to get the message to the right people. This problem is different from a simple network routing problem because

[1]The research was sponsored by the Avionics Laboratory, Wright Research and Development Center Aeronautical Systems Division (AFSC), U. S. Air Force, Wright-Patterson AFB, OH 45433-6543 under Contract F33615-90-C-1465, ARPA Order No. 7597. The first author was sponsored by the National Science Foundation under a graduate fellowship award. 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 or the U.S. government.

[2]The R1/XCON system at Digital is the only production system we know that exceeds 10,000 rules (Barker & O'Connor, 1989). Most production systems still have have only hundreds of productions, though there are some with a few thousand. Systems that have substantially more rules are specialized to have only constants in their patterns (Brainware, 1990), thus avoiding the main source of computational cost, matching of pattern variables.

both communication links and desired recipients are specified in terms of *properties* of people — for example, a communication link might be specified by "John can talk to all the marketing people at headquarters." Also, the data is available only indirectly in a database, which is external to Dispatcher-Soar, rather than part of it. Whenever Dispatcher-Soar needs some piece of information from the database, it must formulate a query using the SQL database-query language, send the query off to a database system, and interpret the database-system's response.

Dispatcher-Soar is implemented using Soar, an integrated problem-solving and learning system, already well-reported in the literature (Laird, Newell, & Rosenbloom, 1987; Rosenbloom et al., 1991). Soar is based on formulating every task as search in *problem spaces*. Each step in this search — the selection of problem spaces, states and operators, plus the immediate application of operators to states to generate new states — is a *decision cycle*. The knowledge necessary to execute a decision cycle is obtained from Soar's knowledge base, implemented as a *production system* (a form of rule-based system). If this knowledge is insufficient to reach a decision, Soar makes use of recursive problem-space search in subgoals to obtain more knowledge. Soar learns by converting this subgoal-based search into *chunks*, productions that immediately produce comparable results under analogous conditions (Laird, Rosenbloom, & Newell, 1986). Chunking is a form of explanation-based learning (EBL) (DeJong & Mooney, 1986; Mitchell, Keller, & Kedar-Cabelli, 1986; Rosenbloom & Laird, 1986).

Dispatcher-Soar is implemented using 20 problem spaces. Figure 1 shows these problem spaces as triangles, arranged in a tree to reflect the structure of the system: one problem space is another's child if the child space is used to solve a subgoal of the parent space. (The boxed numbers will be explained later.) The 8 problem spaces in the upper part of the figure are involved with the dispatching task: three basic methods of performing the task, together with a method for selecting between them. The 12 lower problem spaces are involved with using the external database.

The Experimental Setup

Dispatcher-Soar begins with 1,819 initial (unlearned) productions, and uses a database describing an organization of 61 people. The details of this organization were created by a random generator, weighted to make the organization modestly realistic, e.g., people in the same region are more likely to have a communication link between them than people in different regions.

We generated 200 different *base problems* for Dispatcher-Soar, using a random problem generator with weighted probabilities. (A problem is just an instance of the dispatching task, e.g., "Send this message to everyone involved in marketing for project 2.") We then had Dispatcher-Soar solve each base problem in turn, learning as it went along. Starting from the initial 1,819-production system, Dispatcher-Soar required 3,143 decision cycles to solve the first problem, and built 556 chunks in the process. Then, starting with the initial productions plus the chunks from the first problem, the system solved the second problem. It continued in this fashion, eventually solving problem 200 using the initial productions together with all the chunks it had learned on the first 199 problems.

After solving 200 base problems, the system had learned a total of 10,112 chunks (an average of 506 per problem space), bringing the total number of productions to 11,931. Figure 1, (1) in the box, gives the number of these chunks in each problem space and Figure 1 (2) gives their distribution over the spaces.[3] Approximately 30% of the chunks are search-control rules (the *selection* space), used to choose between the different methods the system has available to it. Another 32% of the chunks help implement various operators (many spaces). The remaining 38% contain memorized database query results (the *memory* and *database-query* spaces), and are based on the memorization technique introduced in (Rosenbloom & Aasman, 1990). Whenever Dispatcher-Soar has to go out to the database to find some piece of information, it memorizes it; if the information is needed again, the database will not have to be consulted. These chunks transfer to other problems — Dispatcher-Soar solved 42 of the 200 base problems without consulting the database. However, it did not memorize the entire database — it still had to consult it during problem 200.

The Chunking Rate

Figure 2 plots the rate at which chunks are built. The horizontal axis in the figure gives both cumulative decision cycles and cumulative production firings. (We will show later that these two numbers are proportional.) The vertical axis gives the number of chunks built. The figure shows that the rate of chunking remains remarkably constant over time — on average, one chunk is built every 6.3 decision cycles. The rate of chunking was indeed expected to stay roughly the same throughout the run (Newell, 1990). However, the constant rate is intriguing because it holds up over a long period of time, despite the different problem spaces employed and the variety of chunks learned (see Figure 1). Even more intriguing, the

[3] The results reported here are for Dispatcher-Soar RUN7, and Soar version 5.2.2 using *all-goals* chunking. Times are user cpu time on a DECstation 5000, excluding Lisp garbage collection time. In addition to the chunks reported here, Soar also builds *internal chunks*, which have only constants (no variables) in their conditions and are excised following the completion of each problem. Internal chunks appear to affect our results only slightly and are ignored here. Two problem spaces build no chunks. The *top* space cannot build chunks because it has no parent. The *send-query-and-get-response* (SQAGR) space builds only internal chunks. An additional problem space is used to wait for results to be returned from the external database. It does no problem solving and is omitted here.

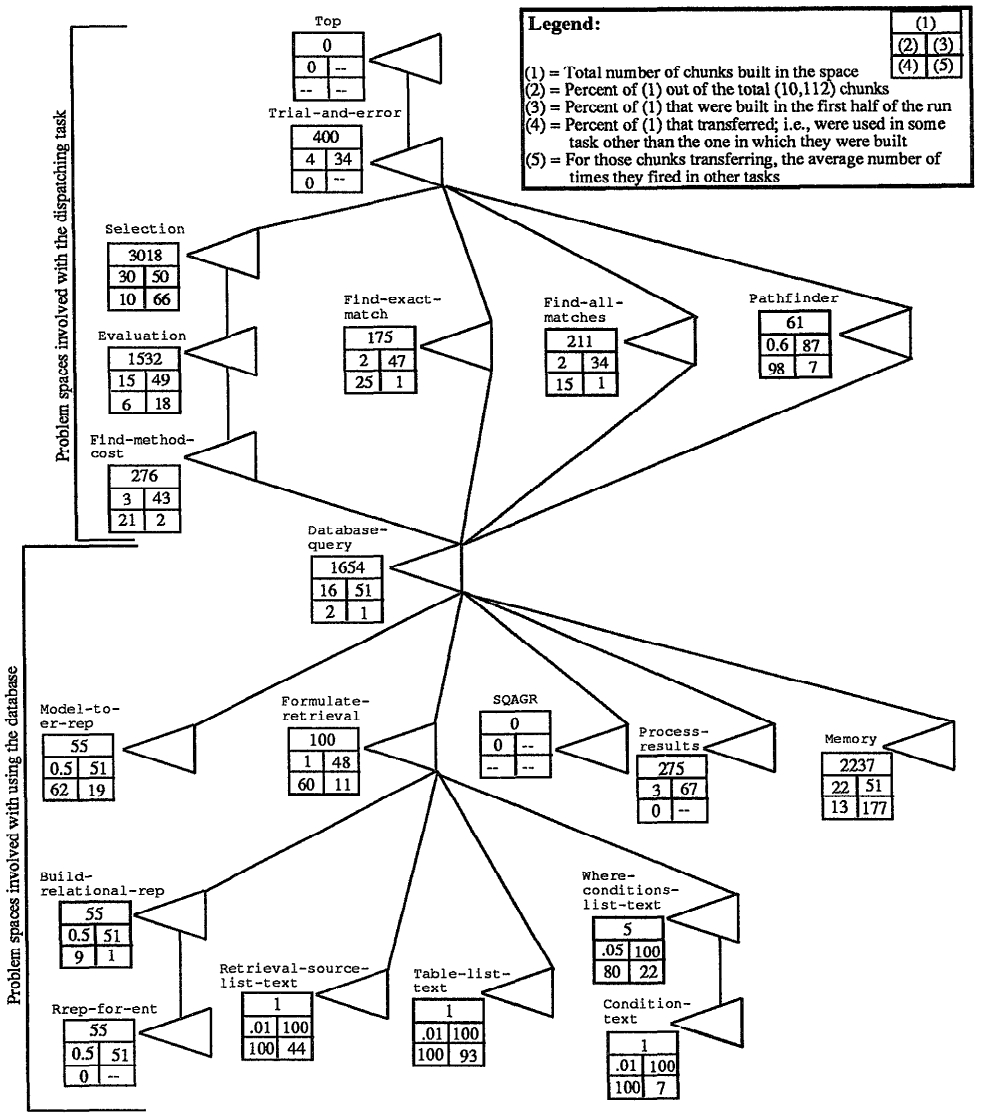

Figure 1: Dispatcher-Soar problem spaces and associated chunks.

same rate of approximately 1 chunk per 6 decision cycles was observed on a smaller scale (~100 chunks) in a completely different set of tasks (Tambe et al., 1990; Tambe, 1991).

A second interesting phenomenon is that in some of the problem spaces, the rate of chunking is highly non-uniform. Figure 1 (3) shows the percent of chunks that were built during the first half of the run, i.e., the first 50% of the total decision cycles required to solve the 200 problems. For example, the figure indicates that 87% of the chunks from the *pathfinder* space were built during the first half of the run. So the rate at which *pathfinder* chunks are built drops significantly over the course of the run. For the *find-all-matches* space, on the other hand, only 34% of the chunks were built during the first half of the run, so the chunking rate for this space is increasing over the course of the run.

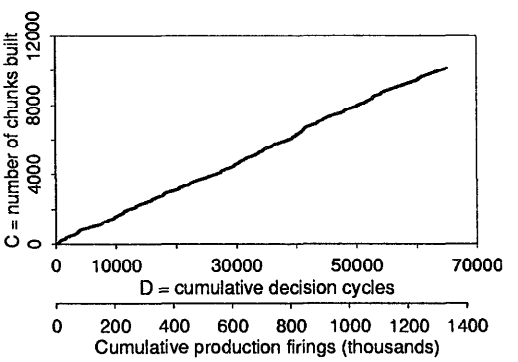

Figure 2: Chunks built on 200 base problems: $C=D/6.3$.

As more and more chunks are built in a problem space, the space may become *inactive*: the space is never used

again, because in every situation where it would previously have been used, some chunk fires to handle the situation instead. In Dispatcher-Soar, some spaces become inactive very quickly: the *table-list-text* space was used during the first problem, built one chunk, and was never needed again — from then on, that single chunk transferred to all the situations where it was needed. Other spaces take longer to become inactive: the *where-conditions-list-text* space became inactive during problem 19, and the *pathfinder* space became inactive during problem 164. Many spaces show no sign of becoming inactive, even after 200 problems.

As mentioned above, Figure 2 has horizontal scales for both cumulative decision cycles and cumulative production firings. In this paper we work primarily in terms of decision cycles, the natural unit of cognitive action in Soar. However, production firings are another familiar base unit. While the number of production firings that make up a decision cycle varies from one decision cycle to the next, Figure 3 shows that the number of cumulative production firings is proportional to the number of cumulative decision cycles, with a conversion factor of approximately 20.5 production firings per decision cycle. Figures 2 and 3 together show that chunks built, cumulative decision cycles, and cumulative production firings are all proportional to one another: 6.3 decision cycles per chunk and 20.5 production firings per decision cycle (hence, 130 production firings per chunk). Our graphs will have extra scales, based on these conversion factors.

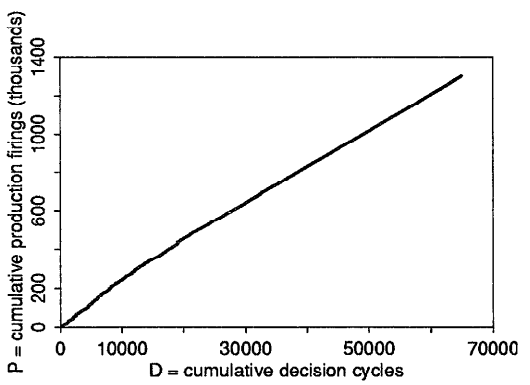

Figure 3: Production firings vs. decision cycles: $P=20.5 \times D$.

The Cost of 10,000 Chunks

Data from existing EBL systems (Etzioni, 1990; Minton, 1988a; Cohen, 1990), including some small Soar systems (Tambe et al., 1990), predicts that a large number of chunks will extract a heavy match cost. This increase in match cost due to the accumulation of a large number of chunks is referred to as the *average growth effect* (AGE).[4] The prediction of a high AGE in Dispatcher-Soar is bolstered by static measurements of the Rete algorithm (Forgy, 1982) employed in Soar for production match: with the addition of 10,000 chunks, the Rete network size (interpreted code size) increases eight-fold.

The mystery that emerges from Dispatcher-Soar is the absence of any average growth effect! Figure 4 illustrates this. It plots the number of cumulative decision cycles over the 200 base problems on the horizontal axis, and the number of token changes per decision cycle on the vertical axis. A token is a partial instantiation of a production, indicating what conditions have matched with what variable bindings. The number of tokens generated by the matcher during a decision cycle is a commonly used implementation-independent measure of the amount of match effort in that decision cycle (Gupta, 1986; Minton, 1988b; Nayak, Gupta, & Rosenbloom, 1988; Tambe et al., 1990). In Figure 4, each point indicates the number of token changes per decision cycle during a 10-decision cycle interval. The figure shows that over the course of about 65,000 decision cycles and the addition of 10,112 chunks, there is no increase in token changes per decision cycle. In fact, a least-squares fit indicates a slightly decreasing trend.

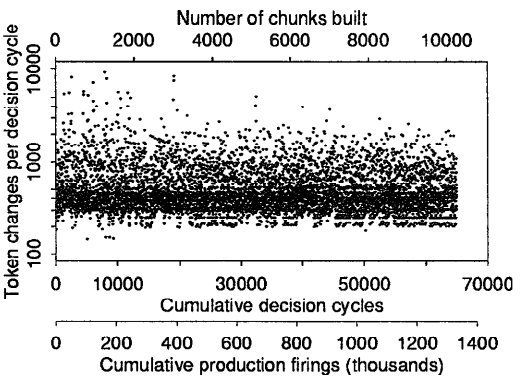

Figure 4: Token changes per decision cycle.

A variety of hypotheses have so far failed to explain this lack of average growth effect. One hypothesis was that the chunks were simply sharing existing parts of the Rete network, thus not adding to the match activity (Tambe et al., 1990). But this is not the case, as the network size turns out to grow in rough proportion to the number of productions in the system. Another hypothesis was that a finer-grained analysis at a level below the decision cycles would reveal the missing average growth effect. Figure 5 presents such a finer-grained analysis: it plots the number of token changes per change to working memory during the 200 base problems.[5] But it shows no AGE either. Yet another hypothesis was that the chunks do not incur match cost, because they don't match

[4]While (Tambe et al., 1990) reported on the AGE phenomenon, their main focus was on *expensive chunks*, i.e., individual chunks that consume a large amount of match effort. Our focus is on the impact of large numbers of individually inexpensive chunks.

[5]As noted earlier, each decision cycle involves about 20 production firings. Each of these involves executing right-hand-side *actions* to add or delete working-memory elements.

variables or because they never match. But the chunks test many variables in their conditions, not just constants. And they often match, as the next section indicates.

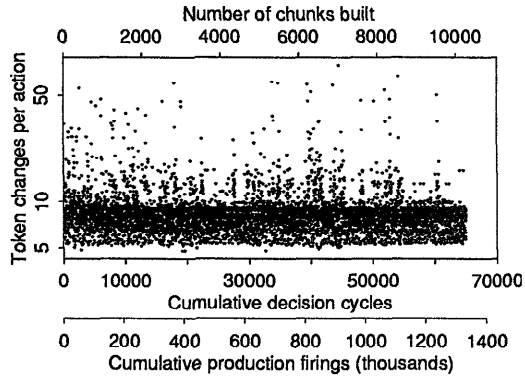

Figure 5: Token changes per action.

Interestingly, the absence of AGE is consonant with the expectations of a different section of the AI community. One of the key experimental results in the area of *parallel production systems* is that the match activity in a production system does not increase with the number of productions (Gupta, 1986; Miranker, 1987; Oflazer, 1987). However, this experimental result is based on non-learning OPS5 production systems (Brownston et al., 1985) with substantially smaller numbers of productions than Dispatcher-Soar (~100 to ~1000 productions). The relation between the results in this paper and those in (Gupta, 1986; Miranker, 1987; Oflazer, 1987) remains unclear.

A second unexpected phenomenon in Dispatcher-Soar is that, in spite of the lack of increase in token changes, the total execution time per decision cycle shows a gradual increase as chunks are added to the system. Figure 6 plots this time per decision cycle over the course of the 200 base problems. Again, each point indicates the time per decision cycle during a 10-decision cycle interval. A linear fit indicates that the time per decision cycle is initially 1.28 seconds, and increases by 0.47 seconds per 10,000 chunks. This is certainly a growth effect, but its source is unclear. The total execution time includes the time spent in matching, chunk building, tracing and other activities of the Soar system. A new, highly instrumented version of Soar is currently under development (Milnes et al., 1992), which should allow us to understand this phenomenon.

The Effectiveness of Learning

To explore the effectiveness of Dispatcher-Soar's learning, Figure 7 plots the number of decision cycles for each of the 200 base problems in sequence. The horizontal axis plots the problem number and the vertical axis plots the problem duration in decision cycles. The figure shows more long problems at the beginning than at the end and that almost all the later problems are short. As Dispatcher-Soar continuously accumulates chunks, it is able to solve problems faster, i.e., in fewer decision cycles, which is direct evidence for effective learning.

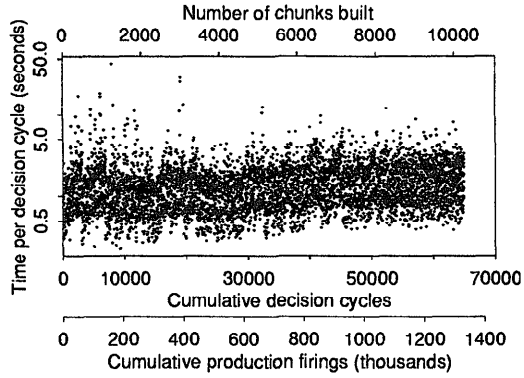

Figure 6: Time per decision cycle.

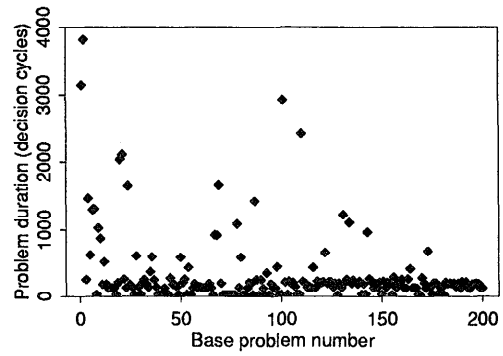

Figure 7: Durations of the base problems.

For a continuously learning system, we can ask whether all the useful learning occurs early on, or whether later learning is still effective. Are the rules learned during the second 10 problems as effective as those learned during the first 10 problems? Unfortunately, a fine-grained analysis of Figure 7 is useless because of the variance in the problem durations. (This has two causes: varying difficulty of the base problems and varying amounts of chunk transfer.) We could smooth out the variance by taking an average every 50 problems — if the average problem-solving time of the last 50 problems were less than that of the previous 50, we could conclude that the learning in later problems is useful. But averaging prevents getting fine-grained information on the effectiveness of learning. Averaging 50 problems, we will not get information about the effectiveness of the learning from the first 10 problems as compared to the second 10 problems.

Instead of averaging, we used a set of *probe problems*. We selected 10 probe problems, generated by the same random generator as the base problems, but disjoint from the base problems.[6] To ensure that this small set

[6]All the problems must be different. If Soar were presented with the same problem twice, the second attempt would take just 4 decision cycles by firing a top-level chunk.

adequately covered the space of problems, we selected problems of varying difficulty: 3 easy, 3 medium, and 4 hard. The difficulty of a problem was judged by giving it to the initial Dispatcher-Soar system (no chunks) and seeing how many decision cycles the system needed to solve it.

The system's performance was then tested on each probe problem, starting with the initial productions plus the chunks from a varying number of base problems. In trial 1, we ran each probe problem starting with just the initial 1,819-production system.[7] In trial 2, we ran each probe problem starting with the initial productions plus the chunks from base problem 1. And so on, at selected intervals, up to the last trial, in which we ran each probe problem starting with the initial productions plus all the chunks from the 200 base problems.

Figure 8 shows how long the system took to solve the probe problems, plotted against the trial number. The three lines (top to bottom) represent the average number of decision cycles needed for the hard, medium, and easy probe problems. The number of decision cycles goes down sharply at first. Hence, the chunks learned in the early base problems are very useful. The decision cycles keep decreasing, but less sharply, throughout the trials. Hence, chunks learned in the later base problems are still useful, but not as useful as the ones from earlier base problems.

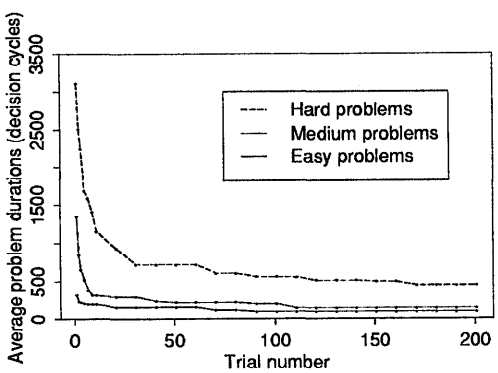

Figure 8: Probe-problem durations, grouped by difficulty.

Figure 9 shows the average number of decision cycles the system needed to solve a probe problem (of any difficulty) plotted against the trial number on a log-log scale. The points fall roughly in a straight line: Dispatcher-Soar's learning follows a power law. This *power law of practice* shows up throughout human skill acquisition, and has been previously observed in Soar (Newell, 1990). That the probe-problem durations follow such a simple rule is remarkable in itself, and strengthens our conviction that the use of probe problems is a crucial technique for the analysis of continuously learning systems.

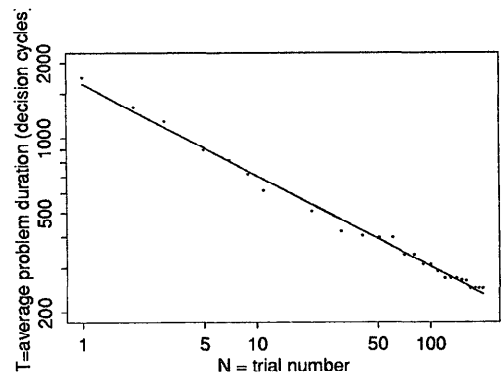

Figure 9: Probe-problem durations, log-log: $T=1683\times N^{-0.365}$.

Taking the ratio of the probe-problem durations in the last and first trials, we see that the overall speedup of the probe problems due to the learning on the 200 base problems was a factor of 7. This gives us an analogue to the *one-shot* learning analysis done in many studies of learning systems (including Soar) (Minton, 1988a; Rosenbloom et.al., 1985), in which the performance of a system is compared before and after learning; i.e., a set of problems is run before any learning takes place; then learning occurs; then the problems are re-run after learning is finished.

Overall, 10.3% of the chunks built during the base problems ever transferred to later base problems. This number seems low at first, but upon further examination it is quite reasonable. First, chunks built in later problems have little opportunity to transfer, since few other problems were run while those chunks were in the system. In fact, the chunks built during problem 200 never get a chance to transfer. Such chunks are hardly useless — with more base problems, the fraction of chunks from the first 200 problems that transfer would increase. Second, some chunks in child spaces never get a chance to transfer, because in all the situations where they would, some chunk in a parent space transfers first and cuts off the subgoaling before the child space arises. The *rrep-for-ent* space is a good example. It built 55 chunks, but none of them ever transferred — for each one, a corresponding chunk from the *build-relational-rep* space always transferred instead.

The effectiveness of learning also varies between problem spaces. Figure 1 (4) shows, for each problem space, what percent of the chunks built from that space in one base problem transferred to some later base problem. For those chunks that did transfer, Figure 1 (5) indicates the average number of transfers per chunk. There is wide variance in the effectiveness of learning by problem space — the percent of chunks that transfer varies all the way from 0% to 100%. Learning was most effective in those spaces where a small number of total chunks were built, but those chunks transferred often.

[7]The system was allowed to learn as it solved a probe problem. However, it was started fresh for each of the 10 probes — none of the other probes' chunks were used. The durations given here exclude a handful of decision cycles spent waiting for the database to respond to queries.

Questions

This investigation is an initial exploration into very large learning systems — systems that acquire a very large number of rules with diverse functionality. The single system used so far, Dispatcher-Soar, has learned over 10,000 rules at the time of this paper. We expect to grow Dispatcher-Soar much further and to grow additional systems. As phenomena and data accumulate, we expect to answer some significant questions about such systems — about what it is like out there. Discovering the right questions is a major part of the enterprise. The general ones posed in the introduction certainly remain relevant. But the results presented here already permit us to formulate some sharper ones.

1. Why is there no average growth effect? Will that remain true further out? Can we have learning systems with 100,000 rules? Does this apply to EBL systems in general?
2. Why does the learning rate stay constant? Is this an architectural constant, independent of task and application system? Will different systems have different (but constant) rates? What is the relationship between the constant total rate and the strongly varying rates by problem space?
3. Does the power law of practice hold further out? Does it apply for different tasks and systems? What is behind it?
4. What is the nature of the rules contributing to early learning (which is very effective) and of those contributing to late learning (which is less effective)?
5. What characterizes spaces that become inactive? How do they affect the mix of rules that are being learned? Can they become active again?
6. What about the structure of Dispatcher-Soar (and other systems and tasks) determines its long-term growth and learning properties?

References

Barker, V., O'Connor, D. 1989. Expert systems for configuration at Digital: XCON and beyond. *Communications of the ACM*, *32*(3), 298-318.

Brainware. Spring 1990. The BMT expert system. Pragmatica: Bulletin of the Inductive Programming Special Interest Group (IPISG).

Brownston, L., Farrell, R., Kant, E. and Martin, N. 1985. *Programming expert systems in OPS5: An introduction to rule-based programming*. Reading, Massachusetts: Addison-Wesley.

Cohen, W. W. 1990. Learning Approximate Control Rules of High Utility. *Proceedings of the Sixth International Conference on Machine Learning*, 268-276 .

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

Etzioni, O. 1990. *A structural theory of search control*. Ph.D. diss., School of Computer Science, Carnegie Mellon University.

Forgy, C. L. 1982. Rete: A fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence*, *19*(1), 17-37.

Gupta, A. 1986. *Parallelism in production systems*. Ph.D. diss., Computer Science Department, Carnegie Mellon University. Also a book, Morgan Kaufmann, (1987).

Laird, J. E., Newell, A. and Rosenbloom, P. S. 1987. Soar: An architecture for general intelligence. *Artificial Intelligence*, *33*(1), 1-64.

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

Milnes, B.G., Pelton, G., Hucka, M., and the Soar6 Group. 1992. Soar6 specification in Z. Soar project, Carnegie Mellon University, Unpublished.

Minton, S. 1988. Quantitative results concerning the utility of explanation-based learning. *Proceedings of the National Conference on Artificial Intelligence*, 564-569 .

Minton, S. 1988. *Learning Effective Search Control Knowledge: An explanation-based approach*. Ph.D. diss., Computer Science Department, Carnegie Mellon University.

Miranker, D. P. 1987. Treat: A better match algorithm for AI production systems. *Proceedings of the Sixth National Conference on Artificial Intelligence*, 42-47 .

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

Nayak, P., Gupta, A. and Rosenbloom, P. 1988. Comparison of the Rete and Treat production matchers for Soar (A summary). *Proceedings of the Seventh National Conference on Artificial Intelligence*, 693-698 .

Newell, A. 1990. *Unified Theories of Cognition*. Cambridge, Massachusetts: Harvard University Press.

Oflazer, K. 1987. *Partitioning in Parallel Processing of Production Systems*. Ph.D. diss., Computer Science Department, Carnegie Mellon University.

Rosenbloom, P.S. and Aasman J. 1990. Knowledge level and inductive uses of chunking (EBL). *Proceedings of the National Conference on Artificial Intelligence*, 821-827 .

Rosenbloom, P. S. and Laird, J. E. 1986. Mapping explanation-based generalization onto Soar. *Proceedings of the Fifth National Conference on Artificial Intelligence*, 561-567 .

Rosenbloom, P. S., Laird, J. E., Newell, A., and McCarl, R. 1991. A preliminary analysis of the Soar architecture as a basis for general intelligence. *Artificial Intelligence*, *47*(1-3), 289-325.

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

Tambe, M. 1991. *Eliminating combinatorics from production match*. Ph.D. diss., School of Computer Science, Carnegie Mellon University.

Tambe, M., Newell, A., and Rosenbloom, P. S. 1990. The problem of expensive chunks and its solution by restricting expressiveness. *Machine Learning*, *5*(3), 299-348.