CHAPTER 21

Placing Soar on the Connection Machine

R. Flynn, Digital Equipment Corporation

1 Introduction

This document is a description of work in progress placing Soar on the Connection Machine. Soar is a production system that has been proposed as an Architecture for General Intelligence[7]; the Connection Machine is a massively parallel fine-grained computation engine[5]. Although the work is still in the early stages, I believe that some conclusions can already be drawn about the appropriateness of the match. In particular, I will try to show that the implementation on the Connection Machine can be made acceptably fast - and will define what I mean by the latter. However, with current production systems, a 10-MIPs uniprocessor workstation would be faster. I will discuss the types of systems for which this situation could potentially change in the future.

2 Why Soar?

The production language that defines Soar is based on OPS-5[6]. It inherits from OPS-5 a working-memory which is matched by productions containing variables, as well as the ability to have productions generate working-memory elements containing new symbols. The problem of matching productions in Soar is therefore very similar to the match in OPS-5; in fact, most of the code is inherited directly from the Lisp Rete match written by Lanny Forgy[2]. Therefore, much of what I describe as implementation issues for Soar on the Connection Machine would apply to most forward-chaining production interpreters.

The differences between Soar and OPS-5 (for the purposes of this discussion) are[6]:

*Connection Machine is a trademark of Thinking Machines, Inc.

1. Soar incorporates a learning mechanism. The learning mechanism allows it to acquire new productions while it is running.

2. Soar does not have a conflict-resolution step between production matching and production firing. Productions can fire in parallel. There are also other opportunities for parallelism in Soar.

3. The syntax of Soar is slightly more powerful than OPS-5. It allows any number of values for a specific atribute of a working-memory element. It also allows for "conjunctive negations," a conjunction of working-memory elements that are asserted not to be in working-memory for the production to match.

In Soar, the processing is broken down into *decision cycles*. Each decision cycle in turn consists of an *elaboration phase* and the *decision procedure*. The elaboration phase is when productions fire. The productions only add elements to working-memory. Sometimes the productions fire in parallel, sometimes a working-memory element added by one production allows another production to fire. The elaboration phase terminates when productions stop firing.

The decision procedure makes a single new decision on the basis of the information provided it by the elaboration phase. The decision made might be setting up a new goal, deciding on a representation for a problem (choosing a problem-space), deciding on the current state, or deciding which operator to apply to the current state. The decision once it is made typically allows another set of productions to fire in the next decision cycle. The decision is effected by attaching working-memory elements to a *context* — the context therefore specifies the currently chosen representation, the current state, and the current operator. Productions generally require the appropriate context to fire. For a much more complete description of Soar, see [7].

There is a second aspect of Soar, not as it currently stands, but as it will evolve, which will allow for much more parallelism. There is a plan to embed the current architecture of Soar in a larger system that incorporates Sensory encoding and Motor Control. In particular, the idea is that working-memory includes sensorimotor information (in some symbolically encoded form), and that independent production systems exist that massage the Sensory and Motor Control information in parallel with the regular decision procedure and context stack. These production systems would not refer to the context stack, and therefore would not have the seriality of the decision procedure imposed on them.

In summary, Soar is an appropriate system to consider for implementation on the Connection Machine because it is interesting, and because it has implicit in its architecture the potential for a high degree of parallelism.

3 Why the Connection Machine?

At first glance, production systems allow for a considerable amount of parallelism. Any production can match at any point in time — all the productions must be considered every decision cycle. However, many experiments with parallelism in production systems have met with limited success. The typical characteristics of current production systems are[4]:

1. working-memory changes slowly.
2. only a small fraction of the total productions apply to working-memory at any moment in time.

The Rete algorithm has been designed to take advantage of these characteristics. It serializes the condition tests in a production so that later conditions are only considered if the prior conditions have been satisfied. Since few productions match any particular situation, with the right ordering of conditions most of the match failures are weeded out early. In addition, Rete saves state about the partial matches across match cycles; it attaches memories to the results of condition tests so that a new working memory when created need only be tested against subsequent conditions. The memory-saving is effective because working-memory changes slowly.

The Rete algorithm has proven very successful in practice. Efficient implementations of it are hard to beat, as we shall see in this paper. In addition, parallel versions of it are beaten by serial implementations. Gupta's thesis argues that this is because the amount of parallelism inherent in production systems (when we take into account the reduction in cost implied by a state-saving algorithm like Rete) is small — speed-ups of the order of 20-40 are achievable on a system with shared memory and 64-128 MIMD processors.

However, the architecture of the Connection Machine is so different that it requires radically different algorithms. Part of the justification for the activity recorded in this paper has been an exploration of the difference, to see if the constraints implied by Gupta's thesis apply[4].

There is a more speculative reason for looking at the Connection Machine. In his talk on "Scale Counts in Cognition" [10], Allen Newell has broken down human cognition into a series of levels of activity - one (neurons firing) at the 1ms. level, one at the 10ms., and one at the 100ms. level. He has also described how Soar would be mapped onto this activity [11]; productions firing at a rate of 10ms., decisions being made every 100ms., and problem-solving episodes completing every second or so. If Soar does map appropriately into Cognition, these rates would seem to indicate that the power of cognition arises from a high degree of parallelism - in the firing of productions, and perhaps in the size of the condition set or the action set of any particular production.

Current Soar systems conform to the constraints of slow memory change, and a small fraction of the productions matching at any moment in time. However, with the advent of Soar I/O (adding sensorimotor architecture), working memory may become highly volatile, and relatively more productions could be firing at any moment in time. Current Soar implementations, both on serial computers, and on a parallel computer such as the Encore Multimax would slow down considerably.

The base rate of communication in the Connection Machine is, highly approximately, 1 ms. (16-bit values can be communicated between all the processors at this rate - with no collisions). The Connection Machine contains relatively slow processors, but the potential for massive parallelism. Therefore, I have been interested in seeing whether the Connection Machine architecture could support a match cycle of 10ms. irrespective of the load or the size of productions. If so, I would like to know whether the opportunities for parallelism promised in the Soar architecture could really provide the leverage required to solve difficult cognitive problems in real time.

4 Implementing a prototype production match mechanism

My first attempt at building a match mechanism was to have a micro-production matcher. Each micro-production consisted of two working memory elements, and one action. In Soar, working-memory elements are predominantly of the form

```
(class <id> ^attr <val>)
```

where `class` and `^attr` can have any name, and `<id>` and `<val>` are generated values that are usually shared with other wme's. A micro-production might be of the form

```
(sp find-the-grandfather
 (human <h> ^father <f>)
 (human <f> ^father <gf>)
 -->
 (human <h> ^grandfather <gf>))
```

The basic intuitive notion behind each of these micro-productions is that they perform one database join. A few other characteristics of Soar affected the design of the algorithm[6].

- In Soar, working-memory is a set, i.e. no duplicate wme's can exist.
- All of the ways a production matches working-memory must be accounted for.

4.1 Connection Machine Operations

In order to understand the subsequent algorithm, a couple aspects of the Connection Machine (hereafter referred to as the "CM") need to be understood[8, 13].

- The CM has a logical addressing scheme where each processor is assigned a number from 0 to 65,535. Global communication functions either *send data* from a local processor's memory to a remote processor's memory, specifying the remote address by number, and location within the remote memory, or *get data* (the reverse). The addressing scheme is logical; since memory reference is by number, almost any type of communications structure can be set up dynamically by performing small computations on each of the processors.

- Almost all operations on the CM can be done with a subset of the total processors, where the subset can be selected by a particular condition. When I speak of "select the processors that," it means set up a condition so that only those processors[1] that will execute the following operation.

- A fast scan operation has been developed which can perform simple associative operations (such as add, multiply, copy) across every processor in the CM, or simultaneously across any number of subsections (where each subsection is contiguous in the logical addressing scheme).

4.2 Elaboration Procedure Algorithm

1. Hash the newly-created wme's to global buckets. The idea here is simultaneously to remove duplicate wme's, and to place wme's in locations such that productions know where to access them. In order to reduce the combinatorics of the match, I broke down wme's by type, where a wme-type consists of the combination of a class name and the attribute name. There is one problem with this: some productions (very few in practice) match with the form

   ```
   (class <id> ^<attr> <val>)
   ```

   i.e. the `^<attr>` can be matched as a variable. For the moment, I have chosen to ignore this difficulty.

   The memory of the Connection Machine is sectioned off into hash tables for each wme-type. The hashing is closed; each table is itself distributed across multiple processors. The hashing algorithm hashes all new wme's in parallel from where the productions created them to their hash table destinations. There are collisions of two sorts: the typical closed hashing to a destination that is already filled, and the possibility that two wme's will try to hash to the same location in parallel. The latter can be resolved arbitrarily in favor of one wme, and collisions can then be handled by re-hashing serially. Since hashing is distributed by wme-type, only the newly created wme's are being hashed, and we can make the hash tables as sparse as the size of the Connection Machine allows, I assume that the number of serial re-hashing steps typically required will be small.

2. Store wme forwarding pointers. The new wme's are sparsely distributed across the hash tables: some way of making them accessible to the productions is required. I assume that the starting location of each wme-type hash table is known to the productions. The productions need to refer to the table in such a way that they can get, for example, "the 5th new wme of type x." This can be done by doing a few scan's to enumerate the wme's in the hash tables, and use the enumerations to store a set of contiguous forwarding pointers at the front of each hash table.

3. Instantiate productions. The productions are allocated one to a processor. One production is created for each combination of wme's that it references. So, for example, if there are 5 wme's of type A, and 3 wme's of type B, a production that has type A wme's as its first condition, and type B as its second would instantiate 15 times. The way the productions know how to instantiate themselves is by having a few more scans executed to count the number of elements in each hash-table. The way the productions instantiate themselves is by calculating how many of each is needed, then calculating where they should move to in memory, moving themselves, and doing a scan to copy out each production the appropriate number of times.

4. Match the productions. Each instantiation looks at a different combination of the wme's and tries to match that combination, in parallel.

5. Perform production actions. For those that have successfully matched, perform the actions of the production. One step of this may involve the generation of new symbols (to be matched by subequent variables). The variables themselves are allocated one to a processor. The mechanism to acquire multiple unique variables in parallel is to look at a random processor, see if it is "free," and if so, use its address as the variable (and mark it as no longer free). Since all the processors can do this simultaneously, the presumption is that there won't be many collisions. Once new variables have been acquired, new wme's can be created, and the cycle can resume.

[1] It is worth mentioning that with the CM, one frequently refers to processors and memory (that is local to each processor) interchangeably.

4.3 Conclusions from the simple algorithm

I wrote a production that would elaborate without end, i.e. it added a new wme that it would match on in some new combination in the subsequent cycle. I then ran this fake elaboration cycle and tuned it extensively to see what the performance figures came out as. The figures were, very approximately, 60ms. an elaboration cycle on the CM-1 (the first machine that Thinking Machines built). This figure could degrade to a maximum of about 100ms. or so with a full load, because a few operations would start to have a lot of collisions. Appendix A makes a desk calculation of what the cost of the algorithm would be, if it were maximally tuned, under a full load, which coincidentally arrives at a similar figure.

The CM-2, which I did not try to run the algorithm on, performs standard operations in about 1/2 the time, and handles situations with collisions much faster. The expectation, from the set of estimates in Appendix A, is that these would take about 16ms. So it seems that my goal criterion of $\tilde{1}$0ms. is not at all far off.

10ms. is both a good figure and a bad one. On the one hand, 10ms. was my original "target" for a production match cycle, because of the belief that this was a plausible rate for human performance. On the other hand, only a micro-production (a single join) was being performed, and current Soar productions typically have as many as 40 conditions, (and some have been known to have 140 conditions ...). Setting up a test of the conditions as a log tree makes the number of serial steps to match a full Soar production 6 - 7, which brings the estimate back up to 60 - 70ms. It is not implausible that improvements in CM technology could bring this figure back down to the $\tilde{1}$0ms. level.

5 Implementing a more complete version of Soar

My next goal was to implement a working version of the problem-solving portion of Soar, i.e. adding the decision procedure, but not including the chunking mechanism. At the same time I have considered different match mechanisms. The coding of an implementation is not complete. One version of a matcher has been written, but it encounters combinatorial difficulties, as we shall see. No code has been written for the decision cycle, but a detailed design exists (the decision procedure design is, at the moment, orthogonal to the mechanisms of matching). To make a workable implementation, a fair amount of front-end code would have to be written (to parse the productions, assign wme-types to locations, and translate between front-end and back-end representations for tracing operations). No design for this exists, although I believe it falls fairly naturally out of the implementations of the above mechanisms.

What I will concentrate on here is a description of the algorithm I wrote for the new matcher. The general consensus is that production matching is the main computational cost of every production system.

5.1 A more parallel matcher

I will present here the incremental changes to the first matching mechanism in the order that they were considered.

Multiway Join The general idea here is that the log tree involved in doing binary joins in the first algorithm is unnecessarily serial. A preferable way to handle things, on the CM, is to do a multiway join. In particular, the join should be as large as is necessary for each production that fires to do so in one step. We would then have a match mechanism that will match productions in constant time irrespective of the productions' sizes.

So, the first step is to imagine a production distributed over multiple processors on the CM. There are separate production nodes, some for constant test conditions, some for match (join) tests, some for actions. Of these, some nodes are specified as "input" nodes, which receive a working-memory element, and may distribute it to other nodes that test (or perform actions) on the same node. Matching a production involves performing all the tests in parallel, and then using a scan *and* operation to see if all the conditions were successfully met.

There are two complexities. The first is setting up the productions so that the right number of instantiations of each production can be created, and then copying the productions out correctly. Recall that in the previous algorithm the instantiations are calculated by examining the number of wme's of each type the production looks at, and multiplying these numbers together. The same holds true here, except the multiplication is n-way instead of two-way, where n is the number of input nodes the production has. The multiplication is handled by — you guessed it — another segmented scan operation. In order to copy out the productions, the individual production nodes are distributed so that they are no longer contiguous (the number of free processors between each node being the number of production instantiations wanted), the scan *copy* is done, and then the nodes re-group themselves so that each production is contiguous again.

The second complexity is determining how to enumerate the productions so that each one refers to a different combination of the working-memory elements. See the design document for a detailed description of this - computationally the algorithm involves a few scans, a divide and a modulo operation.

Saving State The Rete algorithm saves state for partial matches. The first algorithm I described saves state in the form of temporary wme's. Inspired

by a description from Anoop Gupta of Kemal Oflazer's state-saving algorithm [4, 12], I developed a state-saving algorithm similar to Oflazer's. The general idea is that productions try to match even when there is an incomplete set of working-memory elements for them to match against. In particular, the productions have a null wme that matches against everything. Productions start off with all null wme's, and gradually acquire real wme's. Only the combinations that successfully match, partially, will survive to the next cycle of additions to working memory. So a weeding out process does happen, not on the basis of the order that conditions are tested, as in Rete, but on the basis of the order that new working-memory elements are created. Alternatives to the state saving algorithm will be explored later.

Conjunctive Negations Conjunctive Negations are an important part of the current Soar repertoire. In order to implement them on the CM, some modifications need to be made to the matching algorithm. The semantics of a conjunctive negation are, "and, if there are no combinations of working-memory elements that satisfy these constraints."

The result of this is that there may be many combinations (or production instantiations) that will work, but hitting one particular combination with one instantiation needs to inhibit all the other instantiations with similar values from firing. It can be done by presuming that all these "similar" instantiations are contiguous in the CM, and that a match of the negation in one production can be propagated by a scan operation across these instantiations to inhibit all of them from working. Similarity can be imposed by requiring that each conjunctive negation be at the end of the conditions of a production. The instantiation mechanism works so that all production instantiations with the same positive conditions will be contiguous in memory. For the moment, I have required that each production have only one conjunctive negation. The same algorithm can be extended to a small fixed number of conjunctive negations (where each negation is considered serially), but I'm not sure how to extend it further.

5.2 Conclusions about Time

Appendix B makes timing estimates for the second algorithm in the same manner as the first. The conclusions are, very roughly, that, for the CM-2, the cost incurred for time is only an extra 6ms., or about a 1/3 again slowdown for a speed of 22ms. There is slightly extra functionality in the second algorithm, because it has included conjunctive negations. So it seems that, given timing considerations alone, the extra cost of proceeding to a state-saving, multiway join is well worth it.

5.3 Conclusions about Space

The purpose of including a state-saving algorithm is to reduce the space combinatorics required to test productions against all combinations of working memory. The theory behind saving state is that computation can happen on working-memory elements when they are created, without waiting for a complete combination that could potentially match the production. The pre-computation avoids repetitive computation later, and should considerably reduce the set of alternatives that are considered.

However, the algorithm as I have implemented it has space problems that preclude it from being used in its current form. Its premise is that new working memory elements when created only replace the null wme's in some partial instantiations. In order to make this possible, all combinations of null wme's with real wme's that have survived a partial instantiation need to be kept around for a particular production. The set of combinations increases as a production gets closer to a complete match. So, for example, a production with 40 conditions (large, but known in Soar) which matched would require $2^{40} - 1$ partial instantiations (the power set of the conditions).

Many of these partial instantiations might be shared by other "partial matches." So it is possible that the state-saving algorithm would be effective if productions were kept down to 5 or 6 conditions. This would mean that larger productions would have to be serialized in a manner similar to the binary tree for the simpler algorithm. Without some simulation, it is hard, however, to be certain that the state-saving algorithm as I have outlined it is useful.

Furthermore, I have no conclusions about the amount of partial matching that would go on for situations that never led to successes. Again, simulations are needed to evaluate these problems.

6 Alternative State Saving Match Algorithms

Because of the observations of the last section, some review of the alternatives needs to be made.

Generalising the first algorithm. One solution is to generalise the first algorithm, using the production representation for the second, but no state-saving, and allowing n-way joins for some fixed number n. The depth of the binary tree could then be reduced. Whereas a binary tree could handle 40 conditions in 6 steps (2^6), and a typical number of 10-20 conditions in 4-5 steps, this would be reduced for a 5-way join to 2 steps for the typical cases, and 3 for everything commonly encountered.

The main trade-off in increasing the dimensions of the join is the space required to perform the join. It would seem that increasing the join size would decrease the size of temporary, "state-saving" working-memory.

Combining the two algorithms. Perhaps a better solution is to use fixed-sized productions, as the previous alternative suggests, with a flat logarithmic hierarchy, and incorporate the state-saving algorithm as I described.

Adopting more of Oflazer's strategy. Oflazer's strategy is to remove redundant partial instantiations. His definition of a redundant instantiation is one that contains the null wme in a specific slot, where there is another instantiation that is identical except that that slot is filled with a real wme (and maybe some other null slots are also filled). Just keeping the maximal set of consistent partial instantiations would remove the difficulty that I encountered.

However, there is a problem in the generation of new partial instantiations from the maximal set. In my algorithm, new partial instantiations would always be created by replacing one or more null wme's with real ones. In Oflazer's, new partial instantiations must sometimes be generated by replacing already filled slots with new wme's. His strategy for resolving match failures in these newly generated partial instantiations is to replace the conflicting wme with the null wme. So, partial instantiations can breed both more complete instantiations, and less complete ones. The difficulty arises in detecting duplicates, when they are created, and in detecting instantiations that are not maximal.

The first problem can be handled by a sort and compare step, where the sorted production nodes end up adjacent to each other, and duplicates can be detected. Furthermore the correct ordering of the production instantiations is necessary to get conjunctive negations to work (the correct ordering was automatic with my algorithm). The second problem is harder, and if solved, might encompass the first. Oflazer's mechanism is to broadcast each instantiation serially to all the others — as far as I am concerned, an unacceptable solution for the CM.

A weaker solution is to remove redundancies when a number of instantiations happen to be adjacent, and can be compressed into one "semi-maximal" instantiation. It can't be guaranteed to be a maximal instantiation, because there may be another non-adjacent instantiation that would encompass it. It is possible that the combinatorics of this solution would be good enough to make it viable. Nevertheless, the cost of adding a sort step to the algorithm (currently about 16ms.) almost doubles the length of an elaboration cycle. Simulations need to be made to determine the relative trade-offs of the two algorithms.

7 Alternative Match Algorithms

There is an informal consensus among the Soar community that the current match language for Soar may be too powerful. For a complete contrast, it is worth examining Guy Blelloch's Master's thesis, which implements a production matcher on the Connection Machine. In order to do so, he compiles out all variables, calculates the maximum number of instantiations of each production needed, and converts the whole matching mechanism into a signal-passing network of elementary nodes. Statistics he has provided would indicate that it can probably perform at the 10ms. level for the matching of a Soar-style production.

The reason why a similar compilation mechanism cannot be implemented for Soar is that it acquires new productions. The maximum number of instantiations of any one production cannot be predicted, because the size and configuration of working-memory cannot be predicted.

One possibility, however, is to set up channels for communication of wme's from the output of one production to the input of another, for the most frequently used paths. There could be two types of match, a fast match for certain production configurations which have been compiled, and a slow match of the sort I described above for any configurations that don't conform. The system could even allow for some error in this domain, not always waiting for the slow match to complete.

A number of other possibilities have been discussed. Most fall into the category of serializing the match process by extending it over multiple production firings in one elaboration phase, or even over multiple decision cycles. One type of production, already implemented in Soar 4.4, is described as "serial." It only fires one instantiation on every elaboration cycle; it turns out that this type of operation is necessary to count the number of elements in a set.

8 Comparison with a 10-MIPs uniprocessor workstation

Current observed performance of a Rete-based matcher written in C is about 8.8 ms. per production action on a 0.75 MIPs processor, when running the eight-puzzle (personal communication, Milind Tambe). This figure can most probably be bettered by some factor in a uniprocessor implementation. The eight-puzzle is a small production system that nevertheless incorporates a fair amount of combinatorics — Oflazer's simulation had some difficulties with it. Extending this performance to an hypothetical 10-MIPs processor (assuming that the processors are comparable) would create a rate of .67 ms per production action.

If we assume a 10 ms. cycle time on the CM for real productions, the eight puzzle contains about 4.5 actions per production firing, and fires an average of 3 productions at a time, we get a performance rate of about .8 ms. per production action — slightly slower. This processing rate would decrease correspondingly for slower cycle times on the CM.

These figures are only intended to give a very rough outline of performance

comparisons. Although the amount of parallelism in current Soar production systems might be slightly higher than the eight-puzzle, they will not be an order of magnitude different.

9 Conclusions

I have made a first attempt at implementing Soar on the Connection Machine. Although the implementation is incomplete, I now believe it is possible to build a version of Soar on the CM that will be acceptably fast — not immediately, but with some work. My definition of "acceptably fast" springs from two sources — the first is a model Allen Newell has developed of the rate of human cognition. The other definition springs from what current implementations typically achieve.

On the other hand, the Connection Machine is not cost-effective in comparison with a fast serial processor, on existing production sets. Furthermore, the Connection Machine implementation has a number of problems with space combinatorics.

In order for the Connection Machine to scale to tasks which have a high wme volatility, as has been predicted for some Soar tasks in the future, the space combinatorics problem must be effectively dealt with. The increase in time problems for a serial Rete-based Soar matcher currently translate directly into increased space problems for a Connection Machine algorithm. We need to characterize the space requirements of various CM parallel algorithms before it can be shown that the CM will be appropriate for future Soar applications.[2]

A Timing Estimates for First Algorithm

This appendix makes timing estimates based on the written code for the first algorithm. It is an estimate for the elaboration cycle, and only includes costs for communicating between processors, not for the computations on the processors, because the first dominate in this algorithm.

The estimates are not precisely for what the code contains, because there are some available tuning possibilities. The first is to use two processors for each production, with each processor accessing different wme information for the first condition and the second condition. This reduces the two times references with many collisions are required. The second saving is to avoid referencing the same value twice.

[2]One big advantage of using the CM to learn about the space requirements is its software environment. The environment is Lisp-based, and sophisticated enough so that it may be easier, given a base level implementation of Soar, to write real code for the alternative match mechanisms than it is to build a full-fledged simulation of the problem.

The following numbers are used for estimates. The numbers are partially guesses on my part (esp. for the CM-2), partially gleaned from various sources, including a talk by Clifford Lasser on Advanced *Lisp.

Type of communication	CM-1	CM-2
send no collisions	1.0	0.5
send with overwrite	3.0	1.0
get no collisions	2.0	1.0
get with collisions	4.0	1.5
get many collisions	20.0	2.0
scan	0.5	0.25

Here are the running estimates

locate-new-wmes Two loops, in the inner one,

1 send with collisions to the destination
2 sends no collisions and,
1 get no collisions, to transfer the wme
2 get no collisions to test if the wme's are the same.

The loop might run 1.2 times on the average.

Total 13.0 6.0

store-wme-values 2 scans, 2 sends with no collisions.

Total 3.0 1.5

check-prod-instantiations 1 get with many collisions, 1 send no collisions for the wme-counts.

Total 21.0 3.0

reinstantiate-productions Calculate and copy the productions.

1 scan to find new prod location,
2 sends with no collisions to relocate the prod.,
1 scan to copy the production instances out,
2 scans to propagate the wme-counts

Total 4.0 2.0

match-productions 1 get with many collisions for the wme's

Total 20.0 2.0

perform-production-actions For acquiring new variables, loop, with 1 send with overwrite, and 1 send no collisions. The loop should be executed 1.2 times.

Total 5.0 2.0

Grand Total 66.0 16.5

The estimate is close to the observed because two opposing reasons must cancel out. The first (making the real run faster) is that my test runs were not fully loaded, speeding up the processing by a factor of two, probably, The second (making the estimate faster) was that I assumed that the tuning steps I made would be taken.

B Timing Estimates for the Second Algorithm

locate-new-wmes Same as previous algorithm.

Total 13.0 6.0

store-wme-values Same as previous.

Total 3.0 1.5

instantiate-productions Performs both the tasks of *check-prod-instantiations* and *reinstantiate-productions* in the first algorithm.

1 get many collisions for the wme-count
2 scans to calculate the no. of instances
1 scan and
1 send n.c. to move nodes
1 scan to copy out the nodes
1 scan to enumerate the nodes
1 scan and
1 send n.c. to relocate nodes back together
1 get with many collisions for the wme value
1 send no collisions to propagate adjacent tests
1 scan and
1 get many collisions to propagate new wme internally

Total 66.5 9.5

match-productions 11 scans.

Total 5.5 2.75

perform-production-actions For acquiring new variables, loop, with 1 send with overwrite, and 1 send no collisions. The loop should be executed 1.2 times.

Total 5.0 2.0

Grand Total 92.0 22.0

C Cost Estimate for the Decision Procedure

This is based on the algorithm as specified in [3]. It refers by number to the same steps.

1	1 send no collisions	1.0	0.5
2	1 send n.c.	1.0	0.5
3	1 get n.c.	2.0	1.0
4	1 get many-collisions	20.0	2.0
5	1 sort	16.0	16.0
6	4 scans	2.0	1.0
7	2 scans	1.0	0.5
8	2 scans	1.0	0.5
9	—		
10	1 send :overwrite	3.0	1.0
11	1 get many-collisions	20.0	2.0
12	—		
13	—		
14	—		
	total	67.0	25.0

This estimate does not include the final steps which involve a certain amount of front end activity. It also assumes the typical course of activity, with no impasses or subgoal completions.

References

[1] Guy E. Blelloch. *AFL-1: A Programming Language for Massively Concurrent Computers.* Technical Report 918, Massachusetts Institute of Technology, 1986.

[2] Charles L. Forgy. *On the Efficient Implementation of Production Systems.* PhD Thesis, Carnegie-Mellon University, 1979.

[3] Rex A. Flynn. *Design for Soar on the Connection Machine.* Draft document.

[4] Anoop Gupta. *Parallelism in Production Systems.* PhD Thesis, Carnegie-Mellon University, 1986.

[5] W. Daniel Hillis. *The Connection Machine.* MIT Press, 1985.

[6] John E. Laird. *Soar User's Manual.* XEROX Palo Alto Research Center, 1986.

[7] John E. Laird, Allen Newell, Paul S. Rosenbloom. *Soar: An Architecture for General Intelligence.* CMU Technical Report CMU-CS-86-171, 1986.

[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.