Scaling up Logic-based Truth Maintenance Systems via Fact Garbage Collection

John O. Everett and Kenneth D. Forbus
Qualitative Reasoning Group
The Institute for the Learning Sciences
Northwestern University
Evanston, IL 60201 USA
everett/forbus@ils.nwu.edu

Abstract

Truth maintenance systems provide caches of beliefs and inferences that support explanations and search. Traditionally, the cost of using a TMS is monotonic growth in the size of this cache. In some applications this cost is too high; for example, intelligent learning environments may require students to explore many alternatives, which leads to unacceptable performance. This paper describes an algorithm for fact garbage collection that retains the explanation-generating capabilities of a TMS while eliminating the increased storage overhead. We describe the application context that motivated this work and the properties of applications that benefit from this technique. We present the algorithm, showing how to balance the tradeoff between maintaining a useful cache and reclaiming storage, and analyze its complexity. We demonstrate that this algorithm can eliminate monotonic storage growth, thus making it more practical to field large-scale TMS-based systems.

1. Introduction

Over the past two decades basic research in Artificial Intelligence has resulted in an extraordinary wealth of enabling technologies. How these technologies will scale to address real-world problems has recently become an important new frontier for AI research. Truth maintenance systems (TMSs) [McAllester, 1978;1990], [Doyle, 1979], [McDermott, 1991] in particular hold out the promise of enabling the development of articulate and responsive environments for supporting people in learning, evaluating information, and making complex decisions.

TMSs provide valuable explanatory services to automated reasoners. However, this ability comes at the price of a monotonic increase in the size of the dependency network as new assumptions are added. We have found that this memory demand characteristic can result in performance degradation and even software crashes in fielded applications, when the dependency network expands to fill available memory. This was obviously possible in principle, but we were (unpleasantly) surprised at how quickly it could happen. In one fielded application (described below), the time taken to revise an assumption increased from less than a second for the first change to almost 15 seconds by the thirtieth change, and the application crashed on the forty-eighth. Such times are unacceptable for an intelligent learning environment where rapid response to changing assumptions is essential. While TMSs have been successfully applied to real problems (particularly in diagnosis), we suspect that this problem is preventing more widespread use of TMSs in large-scale applications.

Our solution to this problem is to take the metaphor of a TMS as a cache of inferences more seriously. Caches have update strategies that remove information unlikely to be useful in the future. TMSs already do this partially, by retracting beliefs in response to retracted assumptions. We make the stronger claim, that for many practical applications, it is critical to reduce the size of the dependency network itself in response to changing information. The key idea is to identify a class of facts that are (a) unlikely to be needed again once retracted and (b) almost as easy to rederive via running rules as they would via TMS operations. Deleting such facts and the clauses involving them will eliminate the storage cost associated with parts of the dependency network that are likely to be irrelevant. In essence, the TMS includes a *fact garbage collector* that implements an update strategy for the TMS, viewed as a cache for inferences.

Section 2 begins by describing the motivating application context. Section 3 describes the algorithm in detail. Section 4 analyzes the algorithm's complexity and illustrates its performance empirically. Section 5 discusses related work, and Section 6 summarizes our results and points out avenues of future investigation.

2. The Problem

Intelligent learning environments (ILEs) are a class of applications that make extensive use of the explanatory capabilities of truth maintenance systems. The CyclePad system [Forbus and Whalley, 1994] is an example of an ILE for engineering thermodynamics. CyclePad can be thought of as a conceptual CAD system, handling the mechanics of solving equations so that students can focus on the thermodynamic behavior of a design. CyclePad utilizes qualitative physics to provide explanations that are grounded in natural, qualitative terms. To provide these services, CyclePad uses a customized version of the LTRE, a forward-chaining, pattern-directed inference system described in [Forbus and de Kleer, 1993], which contains a Logic-based truth maintenance system (LTMS) [McAllester, 1978]. All derivations are stored in the

LTMS, and the dependency network so formed provides the grist for a generative hypertext explanation system that lets students explore the consequences of their assumptions.

CyclePad is designed to shift the focus of the student from solving equations to developing better intuitions about thermodynamics. This requires that students explore the consequences of parametric changes to cycles they have designed, often via sensitivity analyses. As a consequence, the underlying LTRE may be called upon to make dozens or even hundreds of changes to assumptions involving numerical values during a typical student session. Each of these changes can have extensive consequences; the analysis of a complex cycle might require between 10 and 30 assumptions, which result in the derivation of hundreds of additional numerical values.

The large number of assume/retract cycles and the potentially large number of consequences affected by each cycle means that the cost of this operation dominates CyclePad's performance. Observations of CyclePad in typical conditions suggest that the amount of memory consumed rises linearly with the number of assume/retract cycles, while the time taken for each cycle rises non-linearly. There are two sources for the non-linear time increase: (1) there are more clauses to process on each iteration, because the dependency network has grown, and (2) as storage space begins to run out, the Lisp garbage collector and the virtual memory system require a much larger fraction of the total time.

3. Fact Garbage Collection

Our solution to this problem is to designate certain classes of facts as eligible for deletion when they become unknown. We preserve the integrity of the dependency network for producing explanations of current beliefs at the cost of greater reliance on the inference engine to rederive facts. This solution makes sense when one can identify classes of facts in which (1) it is unlikely that a particular fact will be believed again once it is retracted and (2) the cost of rederivation is small. Numerical assumptions in CyclePad fit both of these criteria. Using a fact garbage collector and designating statements of numerical values as collectible results in a significant improvement in CyclePad's performance. Empirically, we have assumed and retracted 1000 consecutive values at an average time per assumption/retraction cycle of 3.1 seconds. In contrast, CyclePad with the conventional LTRE on the same example exhausted a 32MB memory space at 48 changes, at which point the average retraction was requiring 46 seconds.

The rest of this section describes the fact garbage collection algorithm, and a particular implementation of it, GC-LTRE. We begin with a brief overview of some relevant LTMS and LTRE concepts, and then describe the fact garbage collection algorithm in detail.

3.1 Overview of LTMS/LTRE

Like all TMSs, the LTMS consists of nodes and clauses, where nodes correspond to facts and clauses enforce belief relationships among those facts. The LTMS encodes clauses in disjunctive normal form. Although this encoding enables clauses to operate in all directions (i.e., not just from antecedent to consequent), any particular use of them in a derivation does identify a set of antecedents and a particular conclusion. Relationships between beliefs are maintained via Boolean constraint propagation (BCP).

The LTRE is a forward-chaining, pattern-directed rule system that uses the LTMS to maintain a database of facts related to one another via logical dependencies. Facts are assigned truth values, which may be *true*, *false*, or *unknown*, according to the label of the corresponding LTMS node. LTRE rules consist of a list of triggers and a body. Each trigger consists of a truth condition and a pattern. The body of a rule is executed for each combination of facts which both match the trigger patterns and whose truth values simultaneously satisfy the truth conditions of the triggers.[1] Once their trigger conditions are satisfied, rules are exhaustively executed in an arbitrary order. The LTMS caches partially executed rules on the node of the trigger fact that failed to match its truth condition, and signals the inference engine to re-queue these rules when that node's label changes appropriately.

Rule bodies interact with the LTMS by asserting new facts and logical relations, which in turn cause the LTMS to add corresponding nodes and clauses to its database. Changes in a fact's label are thus automatically updated by the LTMS, rather than requiring repeated execution of rules. Because of this, the conventional LTRE guarantees that a rule will run exactly once on each set of facts to which it has matched, to avoid the wasted work and duplication of clauses that would result. One subtlety in making a fact garbage collector is ensuring that rules are re-executed in certain circumstances described below.

3.2 The FACT-GC Algorithm

To garbage-collect facts we must (a) be able to identify collectible facts, (b) ensure that all relevant structure is removed, and (c) cache information for restoring clause structure under certain circumstances (discussed below). We address the identification of collectible facts by requiring applications to provide a predicate `GCable?` that is true exactly when a fact is of a type that should be collected when it becomes unknown. (We discuss how to define `GCable?` in Section 3.4.) Ensuring the deletion of all relevant structure and the caching of clause information only requires modifications of the LTMS opera-

[1] The simultaneity requirement is a modification to the LTRE presented in [Forbus & de Kleer, 1993]. As it results in a substantial performance improvement, all versions of the LTRE used in experiments described in this paper have this modification.

tions carried out when retracting an assumption. Here is the new LTMS retraction algorithm:

```
Retract-Assumption(n)
1. Label(n) = UNKNOWN
2. Queue = Propagate-Unknownness(n)
3. Find-Alternate-Support(queue)
4. Fact-GC(queue)
```

The first three steps of `retract-assumption` are identical to those of the LTRE described in [Forbus & de Kleer, 1993], with the exception of retaining the queue computed in `propagate-unknownness` for additional processing. Fact garbage collection is only attempted after seeking alternate support so that still-labeled TMS structure is not removed.

The `fact-gc` algorithm simply finds collectible nodes among those that were retracted and executes `collect-node` on them:

```
Fact-GC(queue)
1. For each node n in queue,
   If label(n) = UNKNOWN and GCable?(n)
   then Collect-Node(n)
```

```
Collect-Node(n)
1. For each clause C that n participates in,
   A. Create-Restoring-Thunk(C)
   B. Delete(C)
2. For each rule instance R which includes n in its
   bound triggers, Delete(R)
3. Delete(n)
```

Step 1 of `collect-node`, the caching of clause information, is somewhat subtle, so we describe it last. Step 2 ensures that every unexecuted rule instantiation formed in part by triggering on `n` is destroyed, both to avoid reintroducing collected nodes and to save the effort of executing moot rules. Step 3 deletes the node itself.

Steps 2 and 3 require processing at both the TMS and inference engine levels. In Step 2, partially-bound rules must be removed from the inference engine and from the TMS's cache of partially-executed rules. This step has the desirable side-effect of a substantial reduction in memory usage, as the number of rules in a conventional CyclePad LTRE can be 2,000 at startup, and rises monotonically to five times that amount during a typical user session. In Step 3, the node and its corresponding inference engine fact are deleted from the TMS and inference engine respectively.

Step 1 caches information necessary to recreate the clause about to be deleted. This is necessary because the traditional contract between inference engine and TMS requires that rules are only run once. Since clauses are the product of rules, the deletion of a clause can cause permanent loss of information from the LTRE unless we take steps to ensure that its creating rule runs again.

An example will clarify this point. Suppose we have the simple database formed by the sequence of transactions:

```
Assume(A)
A ⇒ B
```

where both `A` and `B` are collectible, and `A ⇒ B` is installed by a rule that was triggered by `A` becoming true. Retracting `A` will cause `B` to be retracted, and both will be collected, along with the clause that linked them. This is both desirable and safe; if `A` is assumed again in the future, the same rule will execute to create another clause, and the system will again believe `B` as a result. But suppose that `A` were not collectible. Retracting `A` would cause `B` to be retracted. If we collected `B`, we would delete the clause linking `A` and `B`. But since `A` is still in the database, the rule which triggered on `A` will not be re-executed, and a valid inference is permanently lost.

The subtlety inherent in this step arises from the expressiveness of LTRE rules. Consider the set of literals which participate in the clauses that the execution of a particular rule generates. Generally some subset of these literals (e.g., the antecedents of an implication) will appear in the rule's triggers. Call these *trigger literals*. Our GC algorithm generally requires all non-collectible literals of a rule to be trigger literals, in order to ensure the proper caching of a clause-restoring thunk.

A restoring thunk is needed exactly when a clause is being deleted because its consequent is being gc'd and when at least one of its antecedents is both unknown and not gc-able:

```
Create-Restoring-Thunk(C)
1. Retrieve rule instance R used to create C
2. Find antecedent M in C such that ¬GCable?(M)
   and Unknown(M)
3. If antecedent M found then schedule R to be exe-
   cuted when M receives the appropriate label.
```

The LTMS caches the environment of the rule instance in force with each clause when it is constructed. `Create-restoring-thunk` moves this information from the about-to-be-deleted clause to a cache associated with a non-gc-able and unknown antecedent. In the above example this antecedent would be the node corresponding to `A`. Should `A` become believed at some future time, the TMS will check this cache and signal the inference engine to schedule any rules found there for execution, thus ensuring that the clause implying `B` will be re-instantiated. This is merely an extension of the services that the TMS already provides in keeping track of partially executed rules.

For this algorithm to work correctly, any non-collectible literal which becomes unknown must be a trigger literal. Otherwise, there is no appropriate place to cache the thunk, and step 3 above will not execute. The exception to this is when a rule involves no collectible literals, for this reduces to the standard LTMS case.

There are also two cases in which it is possible that step 3 will not execute yet the LTRE will not permanently lose structure: (a) when the application bypasses the rule engine and installs clauses directly, and (b) when all trigger literals are either unknown and gc-able or known. In the first case, the onus for restoring the clause rests with the application. The only responsibility of the TMS in this instance is to signal the application that a directly-installed clause is to be deleted and return that clause to the application. In the second case the clause will be replaced by a new clause when new gc-able facts corresponding to the trigger literals are inserted into the database because these new facts will match and cause the execution of the rule.

3.3 Applicability to Other Types of TMSs

Although we have implemented the fact-gc algorithm in an LTMS, all of our concepts and algorithms are directly applicable to Justification-based TMSs (JTMS) [de Kleer and Forbus, 1993] as well. It is less clear what the ATMS equivalent of `Fact-GC` would be, given that the ATMS does not retract assumptions. If one added the idea of particular assumptions (or environments) becoming irrelevant, it might be desirable to adapt something like these algorithms for weeding out irrelevant environments and justifications.

3.4 How and When to Use Fact-GC

Fact garbage collection requires that the system designer specify what classes of facts are subject to fact garbage collection, by supplying the procedure `GCable?`. Candidates for collectible kinds of facts are those which

- Constitute a sizable fraction of the LTMS database
- Are unlikely to become valid again once unknown
- Are cheap to rederive

The first constraint determines how much storage fact garbage collection will save. The second constraint concerns the likelihood of needing to re-execute rules, and the third constraint concerns the cost of re-executing rules.

In the case of CyclePad, the two classes of facts we elected to garbage collect were

- (<parameter> `NVALUE` <value>) which states that the continuous parameter <parameter> has as its (numerical) value <value>, a floating point number.
- (<parameter> `PROPOSED-NVALUE` <value>) which states that <value> has been proposed by some rule as being an appropriate value for <parameter>.

The `PROPOSED-NVALUE` statements represent conclusions based on different possible methods for deriving a value, and the `NVALUE` statements represent the particular proposal chosen. (Conflicting `PROPOSED-NVALUE` statements represent a contradiction, since all methods should lead to the same answer.)

These choices satisfy all three constraints:

1. `NVALUE` and `PROPOSED-NVALUE` statements are the overwhelming majority of the facts derived by CyclePad.
2. While users sometimes revisit assumptions, the whole point of sensitivity analyses (a critical activity in building intuition about how thermodynamic cycles work) is to systematically vary input assumptions.
3. Given the off-line compilation of rules and equations into code, the cost of re-executing rules is small.

We suspect that many TMS applications have similar properties.

4. Analysis and Results

We begin by analyzing the complexity of the algorithm, and then illustrate its performance empirically.

4.1 Algorithmic Analysis

We assume that the cost of `GCable?` is negligible. The first three steps are the standard LTMS retraction algorithm, which is worst-case linear in the number of nodes and clauses in the database. The complexity of `Fact-GC` is governed by the size of the queue `Propagate-Unknownness` returns, so we can guarantee that the number of nodes and clauses examined is bounded by the complexity of the normal retraction algorithm. If the operations carried out over each node and clause are of low complexity, then the whole algorithm is worst-case linear.

The operations carried out over each node and clause can be divided into two types, those operations that involve only the LTMS, and those operations that involve both the inference engine and the LTMS. The operations involving only the LTMS are trivial, involving at worst linear operations in the size of the term list of a clause, which is typically very small compared to the number of clauses in the database (e.g., 5 versus 10,000) and so they can be ignored. Those involving the inference-engine can all be implemented as constant-time operations if enough information is cached, or as linear operations (e.g., our current code implements the deletion of rules as a single pass through the entire set of nodes in the LTMS after the rest of the retraction is complete) quite easily. Therefore we conclude that the entire algorithm is linear in the size of the LTMS dependency network.

A more interesting complexity measure is the change in storage required as a function of the number of assume/retract cycles. We assume that the assumption being made is new, in that its statement does not appear in the TMS database until the assumption is made, and once it is retracted, it is never reassumed. In a standard TMS, storage requirements for the dependency network will grow monotonically with the number of such cycles, with the order of growth being linear if there are no interactions between consequences of distinct assumptions. In a fact GC'ing TMS, if the assumptions being made are collectible, there will in the best case be no growth in the size of the dependency network, no matter how many assume/retract cycles occur. However, in many cases there can still be growth, since a non-collectible fact can be introduced as a consequence of a collectible assumption, and such facts are never destroyed. (This growth may assumed to be modest, else one should consider making the non-collectible facts introduced collectible.) As the next section demonstrates, it is possible to approach the best case quite closely in a realistic application.

TABLE 1: Comparative Results of 48 Retract-Assume Cycles: Conventional/GC-LTRE

	Time (seconds)			Facts			
Trial	Execution	GC	Total	GC-able	Total	Clauses	Rules
1	0.73 / 0.63	0.16 / 0.22	0.89 / 0.85	288 / 288	1,014 / 1,014	1,301 / 1,301	2,547 / 2,547
2	0.88 / 0.66	0.27 / 0.27	1.15 / 0.93	441 / 288	1,167 / 1,014	1,629 / 1,301	2,949 / 2,547
3	0.96 / 0.68	0.38 / 0.27	1.34 / 0.95	594 / 288	1,320 / 1,014	1,957 / 1,301	3,351 / 2,547
4	0.94 / 0.63	1.04 / 0.28	1.98 / 0.91	747 / 288	1,473 / 1,014	2,285 / 1,301	3,753 / 2,547
5	0.99 / 0.64	1.27 / 0.28	2.26 / 0.92	900 / 288	1,626 / 1,014	2,613 / 1,301	4,155 / 2,547
6	1.13 / 0.71	1.16 / 0.22	2.29 / 0.93	1,053 / 288	1,779 / 1,014	2,941 / 1,301	4,557 / 2,547
7	1.24 / 0.65	1.81 / 0.27	3.05 / 0.92	1,206 / 288	1,932 / 1,014	3,269 / 1,301	4,959 / 2,547
8	1.35 / 0.69	1.15 / 0.22	2.50 / 0.91	1,359 / 288	2,085 / 1,014	3,597 / 1,301	5,361 / 2,547
9	1.45 / 0.67	2.63 / 0.22	4.08 / 0.89	1,512 / 288	2,238 / 1,014	3,925 / 1,301	5,763 / 2,547
10	1.56 / 0.67	1.65 / 0.22	3.21 / 0.89	1,665 / 288	2,391 / 1,014	4,253 / 1,301	6,165 / 2,547
	:	:	:	:	:	:	:
20	2.73 / 0.66	2.96 / 0.22	5.69 / 0.88	3,195 / 288	3,921 / 1,014	7,533 / 1,301	10,185 / 2,547
	:	:	:	:	:	:	:
30	4.01 / 0.65	10.18 / 0.38	14.19 / 1.03	4,725 / 288	5,451 / 1,014	10,813 / 1,301	14,205 / 2,547
	:	:	:	:	:	:	:
40	5.53 / 0.66	15.18 / 0.22	20.71 / 0.88	6,255 / 288	6,981 / 1,014	14,093 / 1,301	18,225 / 2,547
	:	:	:	:	:	:	:
48(a)	11.12 / 0.65	35.13 / 0.44	46.25 / 1.09	7,479 / 288	8,205 / 1,014	16,717 / 1,301	21,441 / 2,547

(a) Conventional LTRE failed after this trial on a machine with 32 MB of RAM

4.2 Empirical Results

We conducted our tests of the fact-gc algorithm on Pentium-based microcomputers equipped with 32MB of RAM. Times reported, therefore, are typical of those a user of the CyclePad system would experience.

Garbage collection provides little or no benefit to an LTRE in which there are few known facts, so one would expect such cases to reveal the additional cost of the GC operations. This cost turns out to be insignificant; in a CyclePad with the bare minimum of assumptions installed to establish the structure of the problem, the assumption and retraction of different numerical values requires 0.04 seconds at the outset from both the conventional and GC LTREs. However, by the fortieth retraction the conventional LTRE requires 0.14 seconds per assumption/retraction cycle, because it has been steadily adding nodes and clauses to the TMS, whereas the GC-LTRE still requires 0.04 seconds, and its LTRE has remained constant in size.

Worst-case assumption/retraction behavior occurs in CyclePad when the cycle is one assumption short of being fully constrained. This is, however, precisely the situation in which a user would find making and retracting assumptions most useful—when doing so provides lots of information. The results we present below therefore focus on this situation.

To generate the data presented in Table 1, we set up a simple refrigerator cycle example in CyclePad and iterated through 48 assume/retract cycles, incrementing the value of a particular numerical assumption each time. We have run the GC-LTRE for as many as 1000 iterations on several different CyclePad examples with no fundamental change in the results, but we cannot do more than 48 it-

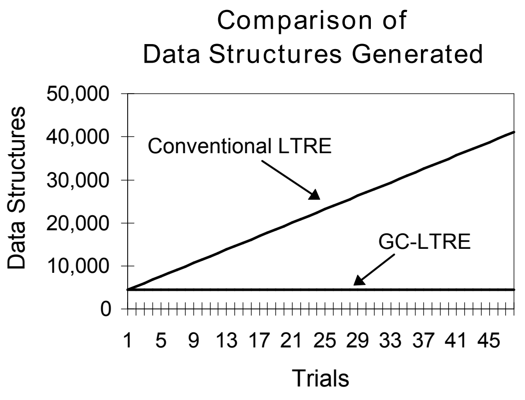

Figure 1

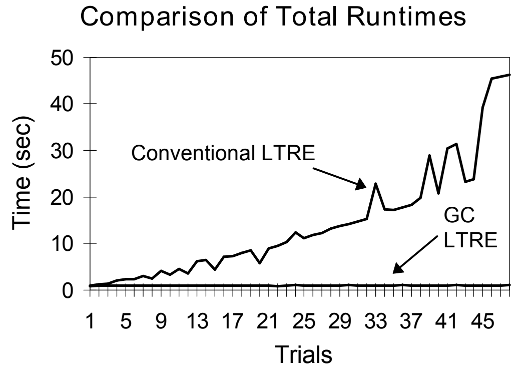

Figure 2

erations in the standard LTRE on the simple refrigerator without causing CyclePad to crash due to an out-of-memory condition.

Table 1 shows the data for both the conventional and GC LTREs. Note that both execution and gc times for the conventional LTRE rise steadily. By the time the conventional LTRE exhausts memory (at 48 assumption/retraction cycles in this particular experiment), data structures have grown by an order of magnitude, and total processing time by two orders of magnitude. Also note that this growth in data structures is evenly distributed among nodes, clauses, and rules.

In contrast we find virtually no growth in data structures in the GC-LTRE, and total time remains virtually constant. The graph in Figure 1 compares the growth in total data structures. In Figure 2, note that the response times of the conventional LTRE are distinctly non-linear. These results indicate that one must be careful to consider the ecological effects of an algorithm as well as its inherent complexity, especially when attempting to scale-up a system.

5. Related Work

We are not aware of any previous paper describing TMS algorithms that reclaim nodes and clauses, nor do we know of any unpublished systems that do so. This may seem surprising at first, but recall that the permanence of the cache provided by a TMS is considered to be one of its defining features. We were only driven to this change by a pressing problem that arose while trying to field software. The term “fact garbage collector” was used by Stallman and Sussman [1977] for one of the first truth-maintenance systems, but their program, like all subsequent TMSs to our knowledge, maintained nodes and clauses indefinitely.

Making TMSs more efficient was a cottage industry in the late 1980s, with most of the attention focused on the Assumption-based TMS (ATMS). The primary advantage of the ATMS is its ability to rapidly switch among many different contexts, but this comes at the cost of an exponential node-label updating process. To address this inefficiency, Dixon and de Kleer [1988] implemented the ATMS algorithm in a massively parallel fashion. Forbus and de Kleer [1988] introduced the implied-by strategy, which enables the ATMS to explore infinite problem spaces by ensuring that the inference engine maintains firm control of the subspace the TMS is searching, although it does not directly address the label-updating problem. Dressler and Farquhar [1991] introduced a means of enabling the inference engine to exercise both local and global control over the consumers of an ATMS. Collins and DeCoste [1991] introduced a modification to the ATMS algorithm which compresses labels into single assumptions, while Koff, Flann, and Dietterich [1988] implemented a specialized version of the ATMS that efficiently computes equivalence relations in multiple contexts. None of these schemes attempted to reclaim storage.

LTMS-based research has focused on finding a practical means of making BCP logically complete for some situations. The issue here is that BCP on individual clauses does not infer all possible implications because logical formulas may require encoding as multiple clauses, and there is no explicit connection among these clauses within the conventional LTMS. Based on the work of Tison [1969], de Kleer [1990, 1992] has developed an algorithm for enabling BCP to operate at the formula rather than the clause level by utilizing prime implicates. In practice we have found that the relative efficiency of clausal BCP is essential in large-scale systems, and we rely on the inference engine to redress the incompleteness of the LTMS.

Of perhaps more interest in this context is the comparison to OPS5 type production systems [e.g., Brownston et al, 1985], which as a matter of course delete facts from working memory. The marriage of such systems with TMSs has not been particularly successful nor widespread. Morgue and Chehire [1991] have combined such a rule engine with an ATMS, and report several problems in controlling the instantiation of facts which are subsequently discovered to be inconsistent, provoking a costly label-updating process.

In contrast to the forward-chaining, pattern-directed inference system we employ, an OPS5 rule system does not run all queued rules, but instead employs either weak methods or domain-specific heuristics to select rules for firing. We believe that, for our task domains, the additional demands that this imposes on the rule author outweighs the additional control afforded.

These reservations aside, OPS5 systems have been shown to scale. Doorenbos has developed an OPS-5 production system that scales to encompass on the order of 100,000 rules [Doorenbos, 1993]. He shows that the Rete and Treat algorithms do not scale well in the matching phase. The solution presented is to augment the Rete algorithm’s shared structure efficiencies (which occur mostly at the top of the beta memory network) with a strategy of unlinking elements near the bottom of the tree from the alpha memory when they contain no beta memory elements. In lieu of garbage collecting, this approach emphasizes parsimony in space and allocation of processing time.

6. Conclusion

Experience with fielding software in an AI application forced us to go back and reexamine one of the fundamentals of truth-maintenance systems, namely that dependency networks should grow monotonically over time. If we take the often-used metaphor of TMS as an inference cache seriously, it stands to reason that we must provide an update strategy for that cache, eliminating elements of it that are unlikely to be useful. The fact garbage collection algorithm presented here provides such a strategy. It is successful enough that we have incorporated it into the current release of CyclePad, which is in use at three universities on an experimental basis by engineering students.

To be beneficial, fact garbage collection requires that a TMS application identify a class of facts that are commonly made as assumptions, that comprise a substantial fraction of the database, and are inexpensive to rederive. We have demonstrated that fact garbage collection indeed greatly benefits a specific application, an intelligent learning environment. We believe that many TMS applications have similar characteristics.

On the other hand, there may be applications where fact garbage collection provides minimal benefit or even harm. An application which creates a large cache of nogoods, each representing the result of considerable computation and intended to be heavily used in future computations, is unlikely to benefit from making the facts involved in such nogoods collectible. Applications which switch back and forth between a small number of sets of assumptions many times may be better off with a traditional TMS—if the available memory can support it. However, we suspect that for many applications, and especially those where the purpose of the TMS is to provide explanations rather than to guide search, the fact garbage collection algorithms provided in this paper can be very beneficial. This, however, is an empirical question.

We plan to try the GC-LTRE on other kinds of TMS-based applications, such as coaches for ILEs and qualitative simulators. Another research group at Northwestern University is currently using the LTRE as the basis for an educational simulator of predator-prey relationships [Smith, 1996].

Acknowledgments

This research was supported by the Computer Science Division of the Office of Naval Research and the Applications of Advanced Technologies Program of the National Science Foundation. We thank the anonymous reviewers for several helpful suggestions.

References

[Brownston et al 1985] L. Brownston, R. Farrell, E. Kant, and N. Martin. Programming Expert Systems in OPS5; An Introduction to Rule-Based Programming. Reading, MA: Addison-Wesley.

[Collins and DeCoste, 1991] J.W. Collins and D. DeCoste. CATMS: An ATMS Which Avoids Label Explosions. Proceedings of the 9th National Conference on Artificial Intelligence, pp. 281-287.

[Dixon and de Kleer, 1988] M. Dixon and J. de Kleer. Massively Parallel Assumption-based Truth Maintenance. Proceedings of the 6th National Conference on Artificial Intelligence, pp. 182-187.

[de Kleer, 1990] J. de Kleer. Exploiting Locality in a TMS. Proceedings of the 8th National Conference on Artificial Intelligence, pp. 264-271.

[de Kleer, 1992] J. de Kleer. An Improved Incremental Algorithm for Generating Prime Implicates. Proceedings of the 10th National Conference on Artificial Intelligence, pp. 780-785.

[Doorenbos, 1993] R.B. Doorenbos. Matching 100,000 Learned Rules. Proceedings of the 11th National Conference on Artificial Intelligence, pp. 290-296.

[Doyle, 1979] J. Doyle. A Truth Maintenance System. Artificial Intelligence 12, pp. 231-272.

[Dressler and Farquhar, 1991] O. Dressler and A. Farquhar, Putting the Problem Solver Back in the Driver's Seat: Contextual Control Over the ATMS, in Proceedings of the 1990 ECAI Workshop on Truth Maintenance, Springer Verlag, 1991.

[Forbus and de Kleer, 1988] Proceedings of the 6th National Conference on Artificial Intelligence, pp. 182-187.

[Forbus and de Kleer, 1993] K.D. Forbus and J. de Kleer. Building Problem Solvers. MIT Press, Cambridge Massachusetts.

[Forbus and Whalley, 1994] K.D. Forbus and P. Whalley. Using Qualitative Physics to Build Articulate Software for Thermodynamics Education. Proceedings of the 12th National Conference on Artificial Intelligence, pp. 1175-1182.

[Koff, Flann and Dietterich, 1988] C. Koff, N.S. Flann, T.G. Dietterich. An Efficient ATMS for Equivalence Relations. Proceedings of the 6th National Conference on Artificial Intelligence, pp. 182-187.

[McAllester 1978] D.A. McAllester. A Three-Valued Truth Maintenance System, S.B. thesis, Department of Electrical Engineering, Cambridge: M.I.T.

[McAllester 1990] D.A. McAllester. Truth Maintenance. Proceedings of the 8th National Conference on Artificial Intelligence, pp. 1109-1116.

[McDermott, 1991] D. McDermott. A General Framework for Reason Maintenance. Artificial Intelligence 50 pp. 289-329.

[Morgue and Chehire, 1991] G. Morgue and T. Chehire. Efficiency of Production Systems when Coupled with an Assumption-based Truth Maintenance System. Proceedings of the 9th National Conference on Artificial Intelligence, pp. 268-274.

[Smith, 1996] B. K. Smith. Why Dissect a Frog When You Can Simulate a Lion? Abstract, to appear in the Proceedings of the 13th National Conference on Artificial Intelligence.

[Stallman and Sussman, 1977] G.J. Sussman and R.M. Stallman. Forward Reasoning and Dependency-Directed Backtracking in a System for Computer-Aided Circuit Analysis, Artificial Intelligence 9 pp. 135-196.

[Tison, 1967] P. Tison. Generalized Consensus Theory and Application to the Minimization of Boolean Functions. IEEE Transactions on Electronic Computers 4 (August 1967) pp. 446-456.