Massively Parallel Assumption-based Truth Maintenance

Michael Dixon[†]

Xerox PARC
3333 Coyote Hill Rd.
Palo Alto, CA 94304

Computer Science Dept.
Stanford University
Palo Alto, CA 94305

Johan de Kleer

Xerox PARC
3333 Coyote Hill Rd.
Palo Alto, CA 94304

Abstract

De Kleer's Assumption-based Truth Maintenance System (ATMS) is a propositional inference engine designed to simplify the construction of problem solvers that search complex search spaces efficiently. The ATMS has become a key component of many problem solvers, and often the primary consumer of computational resources. Although considerable effort has gone into designing and optimizing the LISP implementation, it now appears to be approaching the performance limitations of serial architectures. In this paper we show how the combination of a conventional serial machine and a massively parallel processor can dramatically speed up the ATMS algorithms, providing a very powerful general purpose architecture for problem solving.

Introduction

Efficiently searching complex search spaces is a common need in AI problem solvers. This efficiency has often been achieved by introducing into the problem solver complex control structures that implicitly represent knowledge about the domain, but such designs are inherently error-prone and inflexible. Instead, the Assumption-based Truth Maintenance System (ATMS) [3] provides a general mechanism for controlling problem solvers by explicitly representing the structure of the search space and the dependencies of the reasoning steps. Since its initial development many problem solvers have been built using the ATMS, for problem domains including qualitative physics, diagnosis, vision, and natural language parsing [2,5,7]. However, the ATMS achieves problem solver efficiency by propositional reasoning about problem solver steps, and for large problems these operations comprise a significant amount of computation themselves. In many cases the ATMS can seriously tax the performance of the Lisp Machines on which the original implementation runs.

Massively parallel computers provide orders of magnitude more computational power than serial machines by connecting thousands or millions of processors with some form of communication network. To make such a machine possible the processors must be kept very simple; typically they operate from a shared instruction stream and provide a very limited instruction set. This leads to the major difficulty with massive parallelism: making good use of such a machine requires structuring the task to be distributed among these processors in such a way that the limited computational power and communication available at each processor are well matched to the operations that need to be

[†]Supported in part by the Natural Sciences and Engineering Research Council of Canada (NSERC) and by a grant from the System Development Foundation.

performed. Where such structure can be found, it often involves very different representations and algorithms from those used on conventional machines.

In this paper we show how the propositional reasoning performed by the ATMS is well suited to massively parallel hardware. By implementing the ATMS on one such machine, the Connection Machine built by Thinking Machines Corporation, we can perform ATMS operations orders of magnitude faster than on the Lisp Machine. Moreover, since this implementation provides a superset of the functionality of the original implementation, existing problem solvers built using the earlier ATMS receive these benefits with no further changes.

We begin by laying out the functions the ATMS performs and their role in problem solving.* We then give a brief description of the Connection Machine, and sketch a series of algorithms for implementing the ATMS on it. We present some analysis of the behavior of these algorithms, and close with a few experimental results and some ideas for further exploration.

An Illustrative Example

We will use the following simple search problem to illustrate definitions and algorithms throughout the paper. This is not a very difficult problem and could be solved by much simpler techniques than the ATMS, but will suffice to show how it is used and how it works. At the end of the paper we will say a bit about how the ATMS performs on much harder problems.

> Mr. X must meet with Art, Betty, and Chris this afternoon. There are three opportunities for meetings: at 1:00, 2:00, and 3:00. He must meet with everyone at least once. Art can't come at 2:00. Mr. X would like to
>
> 1. Meet with Art alone.
> 2. Meet with Art before any meeting with Chris.
> 3. Meet with Betty before any meeting with Chris.
>
> Which of these are possible? Can he arrange that all of them happen? Can he arrange them all without any meetings at 3:00?

Assumption-based Truth Maintenance

The ATMS is a general search-control mechanism that can be coupled with domain-specific problem solvers to solve a wide range of problems. Problem solving becomes a cooperative process: first, the problem solver determines the choices to be made and their immediate consequences, and transmits these to the ATMS. Then the ATMS determines which combinations of choices are consistent and which conclusions they lead to. On the basis of these results the problem solver explores additional consequences of those conclusions, possibly introducing new choices. This cycle repeats until a set of choices is found to satisfy the goal or all combinations are proven contradictory.

The ATMS represents problem states with *assumptions* and *nodes*. Assumptions represent the primitive binary choices; in our example there are nine assumptions, corresponding to the propositions

*Although the ATMS has been described in earlier papers by de Kleer [3,4], our development of the parallel ATMS led us recognize that some aspects of that specification reflected the particular representations used by the serial implementation and were thus inadequate to describe a different implementation. We will note the major differences in footnotes.

"Mr. X meets with *name* at *time*", where *name* is one of Art, Betty, or Chris, and *time* is one of 1:00, 2:00, or 3:00. We will refer to the assumptions as a_1, a_2, a_3 (for meeting with Art at 1:00, 2:00, and 3:00), b_1, b_2, b_3, c_1, c_2, and c_3. Nodes, on the other hand, correspond to propositions whose truth is dependent on the truth of the assumptions; in our example "Mr. X meets with Art alone", "Mr. X meets with Art before any meeting with Chris", and "Mr. X meets with Betty before any meeting with Chris" are all represented by nodes, which we will refer to as n_1, n_2, and n_3 respectively.

Dependency relationships among assumptions and nodes are determined by the problem solver and presented to the ATMS as *justifications*. Justifications represent these dependencies as propositional implications in one of two forms:

$$l_1 \wedge l_2 \wedge \ldots \wedge l_m \rightarrow n$$

$$l_1 \wedge l_2 \wedge \ldots \wedge l_m \rightarrow \bot$$

where n is a node, $\bot$ represents a contradiction, and the l_i are nodes, assumptions, or negated assumptions.* The first form indicates a sufficient condition for the truth of a node, the second indicates an inconsistency.

Thus, for example, we record that Mr. X must meet with Chris at least once as

$$\neg c_1 \wedge \neg c_2 \wedge \neg c_3 \rightarrow \bot \qquad \text{[J1]}$$

($\neg c_1$ denotes the negation of c_1). We record that if Mr. X meets with Betty at 2:00, without meeting Chris at 1:00 or 2:00, he will have met with Betty before any meeting with Chris as

$$b_2 \wedge \neg c_1 \wedge \neg c_2 \rightarrow n_3 \qquad \text{[J2]}$$

We also would like to know if n_1, n_2, and n_3 can be satisfied together; to do this we introduce another node n_4, and the justification

$$n_1 \wedge n_2 \wedge n_3 \rightarrow n_4 \qquad \text{[J3]}$$

In order to appreciate both the strengths and the weaknesses of this approach it is important to understand the difference in perspective between the problem solver and the ATMS. To the problem solver, nodes and assumptions represent propositions in the problem domain; their structure is used by domain-specific inference rules and the results of inference are recorded as justifications. To the ATMS, however, assumptions and nodes are atomic; the only relations among them are the justifications the problem solver has reported so far. This makes the ATMS applicable to a wide range of domains, but requires that all the relevant domain structure be represented with justifications.

To specify the behavior of the ATMS, we need some definitions:

- The *assumption space* is the boolean n-space defined by the set of all assumptions. Each point in the assumption space corresponds to some total assignment of truth values to assumptions. We also look at subspaces of the assumption space, which correspond to partial assignments.
- A point in the assumption space *supports* a node if the truth values of assumptions at that point together with the justifications logically entail the node's truth.
- A point in the assumption space is *consistent* if the truth values of assumptions at that point are consistent with the justifications; if they entail a contradiction, that point is *inconsistent*.

*The sequential ATMS does not implement all instances of negated assumptions; our current implementation handles the general case. Furthermore, this implementation is complete without the hyper-resolution rule used by the previous implementation.

- The *extension* of a node is the subset of the assumption space that supports that node, excluding inconsistent points (which support all nodes).*
- A node is *in* if it is supported by at least one consistent point in the assumption space — i.e., its extension is non-empty. Otherwise, of course, the node is *out*.

In our example the assumption space has 2^9 or 512 points; given just the above justifications J1 and J2, n_3's extension consists of the 32 points at which b_2 and c_3 are True, and c_1 and c_2 are False.

The ATMS performs four basic operations for the problem solver:

- create a new assumption
- create a new node
- record a justification
- return a node's extension

In addition to recording the assumptions, nodes, and justifications, the ATMS maintains an efficient representation of each node's current extension, and of the set of points discovered to be inconsistent. Quickly updating these representations after each operation is the key to any ATMS implementation. Creating a node and returning an extension require no changes. Creating an assumption doubles the assumption space (by adding another dimension), and hence doubles the extensions of each node correspondingly.

Adding a justification can change the extensions in very complex ways. Each justification can be thought of as a constraint on the extensions of the antecedent and consequent nodes: the extension of the consequent must include the intersection of the extensions of its antecedents (for the purposes of this discussion we take the extension of an assumption to be all consistent points at which it is assigned True, the extension of its negation to be the consistent points at which it is assigned False, and the extension of $\perp$ to be the set of all currently inconsistent points). If there is no circularity in the justifications (i.e. the nodes can be ordered so that no justification of a node includes nodes that come after it), the extension of each node is just the union over all its justifications of these constraints; if the justifications are circular the ATMS must find the set of minimal extensions that satisfy the constraints.

To compute the new extensions the ATMS uses a form of constraint relaxation. When a justification is added, a check is made to see if the extension of the consequent already includes the intersection of the extensions of its antecedents. If it does not, the consequent's extension is updated, and each justification in which it is an antecedent must now be recursively checked. These changes may propagate arbitrarily far, but it is easy to show that they must terminate. This algorithm is sketched in Figure 1 below.

Suppose in our example J1 and J3 have been recorded so far. The extensions of all nodes are currently empty (since n_4 is the only one with a justification, and all of its antecedents have empty extensions). The extension of $\perp$ is the 64 points with c_1, c_2, and c_3 False. If J2 is then recorded, the intersection of its antecedents will be the 64 points at which b_2 is True and c_1 and c_2 are False, less

*The sequential implementation was formulated in terms of *labels* and *environments*, a particular representation of extensions.

```
record-justification(j_0) :
    J ← J ∪ {j_0}        — the set of all justifications
    q ← {j_0}            — justifications to be processed
    while q ≠ ∅ do
        choose j ∈ q
        q ← q − {j}
        update-extension(j)
        if node-extension-changed then
            q ← q ∪ {j' ∈ J | conseq(j) ∈ ante(j')}

update-extension(j) :
    node-extension-changed ← False
    e ← ⋂_{a∈ante(j)} extension(a)
    if conseq(j) = ⊥ then
        record-inconsistency(e)
    elseif e ⊈ extension(conseq(j)) then
        node-extension-changed ← True
        extension(conseq(j)) ← extension(conseq(j)) ∪ e
```

Figure 1. Computing extensions by constraint relaxation.

the 32 of those which are inconsistent. These points are added to n_3's extension. We next reexamine J3, to see if more points now belong in n_4's extension (none do).

The operations on extensions are thus:

- compute the intersection of the antecedents' extensions
- determine whether the result is subsumed by the current extension of the consequent
- if it is not, compute the new extension of the consequent from the union of the old extension with the intersection of the antecedents' extensions
- remove a set of points that has been discovered to be inconsistent from the extension of each node
- double the extension of every node when a new assumption is added

Choosing a representation for extensions that allows these large set operations to be performed as quickly as possible is the key to building a fast ATMS. The representation used by the serial implementation was too complex and irregular to be efficiently manipulated by Connection Machine; in the next section we will briefly describe the capabilities of this hardware that must be taken into account in designing a new representation, and in the following section we describe the representation we developed.

The Connection Machine

The Connection Machine (CM) is a massively parallel processor designed and manufactured by Think-

ing Machines Corporation [6]. It consists of from 16K to 64K processors, each with 4K to 64K bits of memory. In addition, each processor can emulate several processors, allowing for example 256K virtual processors on a 64K machine, each with one quarter the memory of a real processor. The processors execute from a single instruction stream produced by a host computer (a Symbolics Lisp Machine or a DEC Vax). The basic operations are

- a general bit-field combine operation
- a very low overhead bit move operation between adjacent processors (for the purposes of this operation the processors are on a two dimensional grid, each adjacent to four others)
- a higher-overhead general bit move operation from each processor to any other processor (destination determined by a memory field), implemented by special purpose routing hardware
- an operation that ORs together one bit from each processor

Although all processors share the instruction stream, not all need execute every instruction. Based on the results of previous computations processors may be individually deactivated and later reactivated, effectively skipping the intervening instructions.

To use the CM a program is run on the host machine that generates a sequence of machine-language type instructions (the instruction set is called PARIS). Some parallel extensions of conventional languages (LISP, C, and FORTRAN) that compile to PARIS-emitting code have been implemented; alternatively programs can be written in conventional languages with explicit calls to emit PARIS instructions as they run (this is how the ATMS is implemented). The CM is treated as a large active memory, where each memory location can store a value and perform simple operations on it.

The CM design is intended to be scalable, so that larger machines can be readily built to handle larger problems. Cost is, however, non-linear due to communications complexity (both router size and wire lengths grow nonlinearly).

Representing Extensions on the CM

We present two representations for extensions on the CM and sketch the necessary algorithms. In the first (which we refer to as algorithm A-1) we associate one processor with each consistent point in assumption space. Each of these processors records its assignment of truth values with one bit per assumption; the remaining processors are temporarily deactivated. Node extensions are represented as a subset of the consistent points, by assigning an additional bit per processor for each node to record whether this point supports the node. Computing intersections and unions and testing subsumption are now single bit operations done in parallel by each active processor, and are thus extremely fast. The extension of a node can be returned to the host machine by retrieving the truth value assignments from each active processor that has the appropriate bit set.

Note that the extension of $\perp$ is only implicitly represented as the complement of the active processors; when points are added to it their processors are deactivated. Creating a new assumption requires a forking operation that doubles the number of active processors: each active processor is matched with an inactive processor, which is then activated. The new processors are initialized from the old ones, and the new assumption is assigned True in each new processor and False in each old

one. Each of these steps can be done in parallel by all the processors involved. (The processor allocation step is a standard CM operation; several algorithms are known [6]. Our current implementation uses a very simple rendezvous algorithm with minimal memory requirements, relying heavily on the router.)

The algorithms for updating extensions and creating a new assumption in this representation scheme are sketched in Figure 2 below. Underlined variables are stored per processor, and operations on them are performed in parallel in each active processor. Other operations are just performed in the host machine. $\underline{TV}$ is an array in each processor of truth values indexed by assumptions and nodes; other per-processor variables are temporaries. The function *find-free*() returns for each processor the address of a different inactive processor, and the notation $[\underline{p}]\underline{var} \leftarrow \underline{exp}$ is used to indicate that the value of $\underline{exp}$ is transmitted to $\underline{var}$ in processor $\underline{p}$ (using the router). The function *new-position* allocates a currently unused position in the $\underline{TV}$ array. Finally, the function *any*($\underline{exp}$) returns True if $\underline{exp}$ is True in any active processor (using the global-OR operation), the procedure *activate*($\underline{p}$) makes $\underline{p}$ active, and the procedure *deactivate*() deactivates every processor on which it runs.

---

update-extension(*antes* $\rightarrow$ *conseq*) :

    node-extension-changed $\leftarrow$ *False*
    $\underline{e} \leftarrow$ *True*
    for $a \in$ *antes* **do**
        $\underline{e} \leftarrow \underline{e} \wedge \underline{TV}[a]$
    if *conseq* $= \perp$ **then**
        if $\underline{e}$ **then** *deactivate*()
    elseif *any*($\underline{e} \wedge \neg \underline{TV}$[*conseq*]) **then**
        node-extension-changed $\leftarrow$ *True*
        $\underline{TV}$[*conseq*] $\leftarrow \underline{TV}$[*conseq*] $\vee \underline{e}$

new-assumption() :

    $a \leftarrow$ *new-position*()
    $\underline{TV}[a] \leftarrow$ *True*
    $\underline{child} \leftarrow$ *find-free*()
    $[\underline{child}]\underline{TV} \leftarrow \underline{TV}$
    $\underline{TV}[a] \leftarrow$ *False*
    activate($\underline{child}$)

Figure 2. Parts of Algorithm A-1.

If we apply this algorithm to our example and begin by creating all nine assumptions, we will have 512 processors. Processing J1 at that point will kill off 64 of them. Processing J2 will then mark 32 of the remaining processors as supporting n_3.

As problem solving proceeds, the size of the active processor set continually changes, doubling with the introduction of new assumptions and decreasing as contradictions are discovered. Since the peak processor requirements determine whether or not a problem will run on a particular machine,

success may be very sensitive to the order in which these operations are performed. (Creating all the assumptions first is the worst possible order.)

Our second representation scheme (algorithm A-2) reduces processor requirements by delaying forking as long as possible, on a per-processor basis. This increases the chances both that contradictions discovered elsewhere will make more processors available, and that a contradiction will be discovered in other choices this processor has already made, thereby eliminating the need to fork at all.

To do this we allow each active processor to represent a subspace of the assumption space, by an assignment to each assumption of True, False, or Both (using two bits per assumption rather than one). The processor subspaces are disjoint, and together include all consistent points (in the worst case this representation scheme degenerates to that of A-1). Node extensions are represented as a union of these subspaces, again with one bit per processor. Creating a new assumption now requires no immediate forking; each active processor merely assigns the new assumption Both. Node extensions are retrieved as before; the subspaces are easily expanded to individual points if desired.

Computing intersections, however, becomes more complex: processors in which the result depends on one or more assumptions currently assigned Both must fork before the result can be represented. Consider our example again. After creating nine assumptions we still have only one processor allocated, with every assumption assigned Both (thus representing all 512 points in the assumption space). After processing J1 this processor would fork into three processors, with assignments

c_1: True	c_2: Both	c_3: Both	(256 points)
c_1: False	c_2: True	c_3: Both	(128 points)
c_1: False	c_2: False	c_3: True	(64 points)

(note how these three subspaces partition the set of consistent points). After processing J2 the last of these would again fork, one half assigning b_2 False and the other assigning b_3 True and supporting n_3.

To process all 14 justifications in our example algorithm A-2 requires only 35 processors, resulting in six points in n_4's extension (each corresponding to a schedule meeting all three conditions). Adding the justification

$$n_4 \wedge \neg a_3 \wedge \neg b_3 \wedge \neg c_3 \rightarrow n_5$$

gives n_5 an empty extension, indicating that there is no way to avoid a meeting at 3:00.

How Many Processors Do We Need?

Two obvious questions at this point are "how many processors will these algorithms require?" and "could we use fewer?" Although the CM has a large number of processors, it is easy to see that these algorithms could need exponentially many processors in the worst case (indeed, such an explosion is almost certainly unavoidable: propositional satisfiability, an NP-complete problem [1], is trivially encoded with one assumption for each variable and one justification for each clause).

We can understand the behavior of these algorithms by noting their correspondence with a very familiar class of algorithms: chronological backtracking. Consider first the following algorithm (B-1) for finding all good points in assumption space, and for each point the nodes it supports. This

algorithm processes a sequence of ATMS operations, occasionally recording its state at backtrack points and later reverting to them to reprocess the succeeding operations. The operations are processed as follows:

create assumption: Assign the assumption the truth value True, and record this as a backtrack point. On backtracking, assign False and try again.

create node: Mark this node unsupported.

record justification: If the antecedent fails because of an assumption's truth value, discard the justification. If it fails because of a currently unsupported node, save it with the node for future reconsideration. If the antecedent of a $\perp$ justification holds, backtrack. If the antecedent of a node justification holds, mark that node supported. If it was previously unsupported, reexamine any justifications saved with it.

When all operations have been processed, a good point in assumption space has been found and the nodes it supports determined. This solution is recorded and the algorithm backtracks to find more. When backtracking is exhausted, all solutions have been found.

The correspondence between B-1 and A-1 is very straightforward. The parallel algorithm processes each operation once, using multiple processors, while the backtracking algorithm may process each operation many times. Furthermore, the number of processors alive when the parallel algorithm begins each operation is exactly the number of times the backtracking algorithm processes that operation, as can be proven through a simple induction argument. A simple corollary of this is that the processor complexity of A-1 is the same as the time complexity of B-1.

Algorithm B-2, the corresponding backtracker for A-2, is like B-1 except that choice points are delayed until a justification depending on them is encountered. The same execution-frequency-to-processor-count correspondence holds between these algorithms as between B-1 and A-1.

Although chronological backtracking is used to solve many problems, more powerful techniques are known. The correspondences between chronological backtracking and our parallel algorithms suggest reexamining these techniques in the context of the parallel ATMS. First, note that there are some important differences between parallel and backtracking algorithms in the consequences of such optimizations. Backtracking programs always benefit when a branch of the search tree is eliminated, but the time required by the additional reasoning needed to determine that it can eliminated must be weighed against the time saved by not searching it. The parallel algorithms, on the other hand, receive no benefit if there are already enough processors available, but when the reduction is needed the time spent is clearly worthwhile. (Note that these tradeoffs are further complicated when we introduce sequentialization techniques that process the search space in pieces determined by the number of processors available, but we will not consider such techniques in this paper. Ultimately any parallel algorithm will have to fall back on such a strategy to deal with arbitrarily large problems, but the complexities and trade-offs need much more investigation).

One class of improvements (dependency-directed backtracking) uses information about the contradiction discovered on one branch to cut off other branches. These are not applicable, since the parallel ATMS is exploring all branches in parallel; when it discovers a contradiction in one branch it will simultaneously discover it in all other branches to which it applies.

More applicable, however, are techniques for changing the order in which justifications are considered. Based on the ideas of boolean constraint propagation [9] we can construct algorithm B-3. Rather than processing the justifications in the order presented, B-3 searches first for justifications that will lead to a contradiction or force the value of an assumption (to avoid a contradiction). Justifications that require forking are delayed as long as possible. On the parallel ATMS we have a corresponding algorithm, A-3, that can broadcast justifications with forking inhibited, so that those processors that would deactivate or force an assumption's truth value do so, while those that would fork do nothing. There is no need to keep track of which processors were blocked from forking; all that is necessary is to note that some were and to record that that justification will have to be rebroadcast at some later time. There are limitations, however: all justifications must be completely processed before we can correctly compute a node's extension.

Results and Prospects

We have implemented a version of A-3 that only resorts to delaying justifications when it runs out of processors, and have run several tests on the Connection Machine, including some large qualitative reasoning programs in which performance limitations of the serial ATMS had been a severe bottleneck. The results are encouraging: as expected, the parallel ATMS runs very quickly. The effective speedup for a given problem depends on how much of the problem solver's time the ATMS consumes. Placing thirteen non-attacking queens on a thirteen by thirteen chess board, a problem requiring minimal problem-solver computation and a lot of ATMS computation, ran seventy times faster on a 16K CM than the fastest sequential implementation on a Symbolics Lisp Machine (60 seconds vs. 4235 seconds, to find 73,712 solutions) [8]. We quickly discovered, however, that even hundreds of thousands of processors are insufficient for many problems, requiring that some combination of parallel and sequential search be used. We have had some success in our initial efforts in this direction, but there is much work still to be done here.

While the CM is a near-ideal machine for developing this sort of algorithm, it is natural to ask how much of the machine is needed; if it could be simplified, more processors could be built for the same cost. As mentioned earlier, the major expense in the current CM design is the complex router system. Although the router makes implementing the parallel processor allocation very straightforward, silicon may be better spent on more processors. One possibility would be to simply connect the processors in an m-dimensional grid (like the CM NEWS grid, but possibly with more dimensions) and then use some form of expanding-wave allocation [6] to match up processors. The memory per processor ratio should also be examined; the current CM arrangement gives each processor considerably more memory than it is likely to need for these algorithms.

Also note that high performance communication throughout the processor pool is not required; although all processors must be able to find another free processor quickly, they never need to communicate with other active processors. In fact, a single host could use several CMs with the assumption space divided among them, each allocating from their own pool of processors. Only when one machine became saturated would it be necessary to shift information to another; load-balancing heuristics would help minimize the frequency with which this needed to be done.

Conclusions

Making explicit the propositional reasoning behind problem solvers can make them simpler, more flexible, and more efficient. By exploiting recent developments in hardware design we can minimize or eliminate the performance penalties that have sometimes offset these benefits in the past. The ATMS appears to match the architecture of the Connection Machine particularly well: the serial host machine performs the more complex but local domain inference steps, while the Connection Machine performs the simpler but global operations necessary to determine consistency and support.

The development of the parallel ATMS has also dramatically demonstrated the degree to which working around the performance limitations of serial machines has complicated otherwise simple algorithms. In order to obtain adequate performance the Lisp Machine implementation uses complex representations and elaborately crafted algorithms. Its development and tuning has taken over a year, and the resulting code is about sixty pages long. The Connection Machine algorithms are much simpler, require three pages of code, and took about a week to develop. In doing so we were also led to a clearer analysis of the ATMS, unencumbered by the complexities of the serial implementation's representation.

Acknowledgements

We thank Thinking Machines for providing us with the facilities to develop and test the algorithms we have described, and in particular Craig Stanfill both for his invaluable assistance in using the Connection Machine and for discussions of the implementation. John Lamping pointed out the correspondence with backtracking, and Jim des Rivières and Susan Newman provided very helpful comments on an early draft.

References

[1] Cook, S., The Complexity of Theorem Proving Procedures. *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, 1971.

[2] D'Ambrosio, B., A Hybrid Approach to Uncertainty. *International Journal of Approximate Reasoning*, to appear.

[3] de Kleer, J., An Assumption-based TMS. *Artificial Intelligence* **28** 127–162, 1986.

[4] de Kleer, J., Extending the ATMS. *Artificial Intelligence* **28** 163–196, 1986.

[5] Forbus, K. D., The Qualitative Process Engine. University of Illinois Technical Report UIUCDCS-R-86-1288, 1986.

[6] Hillis, W. Daniel, *The Connection Machine*. MIT Press, Cambridge, Massachusetts, 1985

[7] Morris, P. H., and Nado, R. A., Representing Actions with an Assumption-based Truth Maintenance System. *Proceedings of the National Conference on Artificial Intelligence*, Seattle, July 1987.

[8] Stanfill, C., Personal communication.

[9] Zabih, R., and McAllester, D., A Rearrangement Search Strategy for Determining Propositional Satisfiability. *Proceedings of the National Conference on Artificial Intelligence*, St. Paul, August 1988.