Truth Maintenance

David McAllester*
Massachusetts Institute of Technology
Artificial Intelligence Laboratory
dam@ai.mit.edu

This paper appeared in AAAI-90.
A postscript electronic source for this paper can be found in ftp.ai.mit.edu:/pub/dam/aaai90.ps.
A bibtex reference can be found in ftp.ai.mit.edu:/pub/dam/dam.bib.

Abstract

General purpose truth maintenance systems have received considerable attention in the past few years. This paper discusses the functionality of truth maintenance systems and compares various existing algorithms. Applications and directions for future research are also discussed.

Introduction

In 1978 Jon Doyle wrote a masters thesis at the MIT AI Laboratory entitled "Truth Maintenance Systems for Problem Solving" [Doyle, 1979]. In this thesis Doyle described an independent module called a truth maintenance system, or TMS, which maintained beliefs for general problem solving systems. In the twelve years since the appearance of Doyle's TMS a large body of literature has accumulated on truth maintenance. The seminal idea appears not to have been any particular technical mechanism but rather the general concept of an independent module for truth (or belief) maintenance.

All truth maintenance systems manipulate proposition symbols and relationships between proposition symbols. I will use the term "Boolean constraint" to mean any Boolean formula built from proposition symbols and standard Boolean connectives such as $\rightarrow$ (implication), $\wedge$ (conjunction), and $\neg$ (negation). A *monotonic* TMS manipulates proposition symbols and Boolean constraints. A *non-monotonic* TMS also allows for "heuristic" or "non-monotonic" relationships between proposition symbols such as "whenever **P** is true **Q** is likely" or "if **P** is true then, unless there is evidence to the contrary, assume **Q**". While the semantics of monotonic truth maintenance systems are quite clear, the semantics of non-monotonic systems has been a focus of considerable research over the past decade and has lead to the development of non-monotonic logics.

Non-monotonic logic is closely related to belief functions, certainty factors, and defaults in type hierarchies. A good introduction to the issues surrounding belief and certainty can be found in [Pearl, 1988]. A discussion of defaults in type hierarchies can be found in [Touretzky, 1986]. Some approaches to the theory of non-monotonic logic can be found in [McCarthy, 1986], [Konolige, 1987], [Gelfond and Lifschitz, 1988], and [Gelfond, 1989].

Having briefly mentioned non-monotonic logic, the remainder of this paper is dedicated exclusively to monotonic truth maintenance systems. There are several reasons for this. First, most of the development in truth maintenance algorithms, and de Kleer's ATMS algorithm in particular [de Kleer, 1986a], concern monotonic systems. Second, most practical applications of truth maintenance systems involve monotonic systems (e.g., qualitative simulation, fault diagnosis, and applications to search). Furthermore, monotonic truth maintenance systems provide a solid foundation upon which to build other kinds of systems — algorithms for monotonic systems can usually be used in non-monotonic systems but the converse does not hold.

This survey begins with a specification of the functionality of monotonic truth maintenance systems. This specification is presented as a set of functions that can be used as a generic interface to most existing systems. Each interface function has a clean non-computational specification. After presenting the interface, I present various implementations. This is followed by a discussion of applications of truth maintenance systems in solving search problems. Finally, there is a brief discussion of current research in the construction of more powerful algorithms.

The Generic TMS Interface

A monotonic TMS is a general facility for manipulating Boolean constraints on proposition symbols. For example, in automobile diagnosis we might want to enforce the constraint that if the spark plug is sparking then

*This work was supported in part by National Science Foundation contract IRI-8819624 and in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-86-k-0124.

the rotor is turning. This constraint has the form $\mathbf{P} \rightarrow \mathbf{Q}$ where **P** and **Q** are proposition symbols that an outside observer can interpret as representations of the statements "the spark plug is sparking" and "the rotor is turning" respectively. Given a set of propositions about automobile engines, a set of constraints on those propositions (such as the above implication), and a set of observations about a particular automobile, a TMS can be used to ask questions about the consequences of the observations.

I will describe the functionality of a (monotonic) TMS by specifying four generic interface functions. A TMS stores a set of Boolean constraints (Boolean formulas). Intuitively, one is only interested in truth assignments that satisfy this stored set of constraints. Because these constraints do not appear explicitly as arguments in most of the interface functions, I will call them "internal constraints". The first interface function, **add-constraint**, adds a constraint to the internal constraint set. Once a constraint has been added it can never be removed. The remaining interface functions manipulate literals — a literal is either a proposition symbol or the negation of a proposition symbol. The second interface function, **follows-from?**, takes two arguments, a literal Φ and a set of literals Σ called a premise set. An application **(follows-from?** Φ Σ**)** can return **yes**, **no**, or **unknown**. If **(follows-from?** Φ Σ**)** returns **yes** then the TMS guarantees that Φ follows from the premise set Σ and the internal constraints. If **(follows-from?** Φ Σ**)** returns **no** then the TMS guarantees that Φ does not follow, i.e., there exists an interpretation satisfying both the internal constraints and Σ in which Φ is false. If the TMS is unable to determine if Φ follows, then **(follows-from?** Φ Σ**)** returns **unknown**. In an automobile diagnosis system the internal constraints consist of facts true of all automobiles, e.g., "if the spark plug is sparking then the rotor is turning", and premise sets consist of observations about particular automobiles.

The third and fourth interface functions compute justifications. If the TMS can determine that Φ follows from the internal constraints and a premise set Σ, then one can ask the TMS to justify this fact, i.e., to produce a "proof" of Φ. There are two interface functions used to generate such proofs: **justifying-literals** and **justifying-constraints**. Both of these functions take two arguments — a literal and a premise set from which that literal can be derived. If Φ is derivable from Σ and the internal constraints, then **(justifying-literals** Φ Σ**)** returns a set of literals and **(justifying-constraints** Φ Σ**)** returns a subset of the internal constraints satisfying the following two conditions.

- Φ follows from the literals in **(justifying-literals** Φ Σ**)** and the constraints in **(justifying-constraints** Φ Σ**)**.
- **(follows-from?** Ψ Σ**)** returns **yes** for each literal Ψ in **(justifying-literals** Φ Σ**)**.

Suppose the internal constraint set includes the constraints $\mathbf{P} \rightarrow \mathbf{Q}$, $(\mathbf{P} \wedge \mathbf{W}) \rightarrow \mathbf{R}$, and $(\mathbf{Q} \wedge \mathbf{R}) \rightarrow \mathbf{S}$. Most truth maintenance systems are able to derive **S** from these constraints and the premise set $\{\mathbf{P}, \mathbf{W}\}$. Most truth maintenance systems also provide the following justifications relative to these constraints and premises.

derived literal	justifying literals	justifying constraints
S	$\{\mathbf{Q}, \mathbf{R}\}$	$\{(\mathbf{Q} \wedge \mathbf{R}) \rightarrow \mathbf{S}\}$
R	$\{\mathbf{P}, \mathbf{W}\}$	$\{(\mathbf{P} \wedge \mathbf{W}) \rightarrow \mathbf{R}\}$
Q	$\{\mathbf{P}\}$	$\{\mathbf{P} \rightarrow \mathbf{Q}\}$

For any given set of internal constraints, premise set, and formula **S** that can be derived from the given constraints and premises, the justification functions can be used to generate a "justification tree" for **S**. The root of the tree is the formula **S** and at each node of the tree the function **justifying-literals** can be used to get children nodes until one reaches members of the premise set. The justifications are required to be non-circular, i.e., if **Q** appears in the justification tree rooted at **P**, then **P** must not appear in the justification tree rooted at **Q**.

Note that all of the justifications in the above table are "local" in the sense that, for each justification, there is only a single justifying constraint. In general, we do not require that justifications be local in this sense. However, virtually all TMS implementations are local in that every derived literal can be justified in terms of other literals and *a single* internal constraint.

Contradiction Handling

The above specification of a generic TMS interface allows for contradictory information to be given to the TMS. Most truth maintenance systems have some way of informing the user that a given premise set is inconsistent with the internal constraints. This can be done by adding a special proposition symbol called **contradiction**. If the TMS is able to determine that a given premise set Σ contradicts the internal constraints, then **(follows-from? contradiction** Σ**)** returns **yes**. Furthermore, if **(follows-from? contradiction** Σ**)** returns **yes** then **(justifying-literals contradiction** Σ**)** and **(justifying-constraints contradiction** Σ**)** return a set of literals and a set of constraints respectively that underlie the contradiction. The justification functions can be used to construct a justification tree whose leaves are literals in Σ. This allows the contradiction to be "blamed" on a subset of Σ.

BCP Implementations

Suppose that we wish to compute a value for an application **(follows-from?** Φ Σ**)**. This can be done with a conceptually simple procedure known as *Boolean constraint propagation* (BCP). Consider a network whose nodes are the proposition symbols that appear either in the premise set Σ or in the internal constraint set. Each Boolean constraint can be viewed as a connection, or "link" between nodes in the network. Each node (proposition symbol) can be labeled with one of three possible labels: **true** , **false**, or **unknown**. Initially all nodes are labeled **unknown**. To compute the consequences of a particular premise set Σ one assigns the label **true** or **false** to each proposition symbol in the set of literals Σ depending on whether that symbol appears positively or negatively in Σ. New labels are then computed based on local propagation — whenever a new truth label follows from existing labels and *some single internal constraint*, that new label is added to the network and propagation continues. If a set of derived labels ever violates one of the internal constraints, then the special proposition **contradiction** is labeled true. During the propagation process each newly derived label can be associated with a "justification", i.e., a data structure that records the labels and constraint used in the derivation. If every constraint is a clause, i.e., a disjunction of literals, then this propagation process can be run to completion in time linear in the total size of the set of constraints [McAllester, 1980]. To answer a query of the form **(follows-from?** Φ Σ**)** one simply runs the Boolean constraint propagation process starting with the labels in Σ and determines if a label is derived for the proposition symbol in the literal Φ — if a label has been derived for the proposition symbol, and that label has the same sign as the sign of the literal Φ, then **follows-from?** returns **yes**, otherwise **follows-from?** returns **unknown**.

Boolean constraint propagation is not logically complete. For example, consider the constraints $\mathbf{P} \rightarrow \mathbf{Q}$ and $\neg\mathbf{P} \rightarrow \mathbf{Q}$. The literal $\mathbf{Q}$ follows from these constraints but does not follow from either constraint individually. BCP will not deduce $\mathbf{Q}$ even though $\mathbf{Q}$ follows from the constraints. The incompleteness of BCP represents a compromise between functionality and efficiency. Boolean constraint propagation runs in linear time in the total size of the constraint set (for clausal constraints). Boolean entailment, however, is coNP-complete and no efficient algorithm can be expected.

Incremental Context Switching. Often consecutive queries to a truth maintenance system have very similar premise sets. For example, a user might first ask for **(follows-from?** Φ Σ**)** and then ask for **(follows-from?** Φ Σ'**)** where Σ and Σ' are large premise sets that differ on only a few literals. The propagation used to answer the first query can be used to make answering the second query more efficient. This can be done with "incremental premise retraction" and "incremental premise addition" [Doyle, 1979], [McAllester, 1980]. Incremental addition and retraction algorithms allow the set of proposition labels to be incrementally switched from the labeling generated by Σ to the labeling generated by Σ'.

An alternative to unrestricted incremental retraction is to store the current premise set on a *premise stack*. When a new premise is pushed on to the premise stack incremental Boolean constraint propagation is used to add new truth labels. When a premise is popped, a simple "undo-list" can be used to remove all labels that were added when that premise was pushed. Now consider two consecutive queries of the form **(follows-from?** Φ Σ**)** and **(follows-from?** Φ Σ'**)**. The labeling for the first query is incrementally computed by pushing the elements of Σ onto the premise stack. To compute the labeling for Σ' one first pops back to a premise set that is a subset of Σ' and then pushes those premises in Σ' that are not already present.

The premise-stack implementation and the unrestricted incremental implementation provide the same generic interface functionality — at the level of the generic interface one can ask about any premise set at any time. Whether the premise-stack implementation is more efficient than the unrestricted incremental retraction implementation depends on the statistics of consecutive queries. For individual retractions the premise stack implementation is considerably more efficient. However, the premise stack implementation may require more retractions and assertions to make a transition between two labeling states. In any application where a set of truth assignments is being systematically explored, such as most algorithms for solving constraint satisfaction problems, the premise stack implementation is more efficient. The premise stack implementation is also more efficient if most premises are static (are always included) and only a few premises are being changed — one can arrange for the changing premises to migrate to the top of the stack so that only small retractions and additions are done when switching between contexts.

ATMS-like Implementations

I will initially describe de Kleer's ATMS [de Kleer, 1986a] as an alternative implementation of the generic TMS interface and compare the ATMS implementation with the BCP implementation. For many applications the only difference between the ATMS and BCP implementations is there relative efficiency as implementations of the generic interface. However, the ATMS is more than just an alternative implementation of the same interface — there are certain additional interface functions that can be easily implemented on top of the ATMS algorithms but not on top of BCP. The additional functionality of the ATMS is discussed below.

Universal Propagation. Like the BCP implementation, the ATMS implementation of the generic interface operates by propagating labels on a network whose nodes are propositions connected by Boolean constraints.[1] However, in the ATMS implementation

[1]The ATMS described here is a recent version [de Kleer,

the propagation process is independent of any particular premise set — a single "universal propagation process" precomputes all answers to all possible queries [de Kleer, 1986a].[2] To make the universal propagation process more efficient, the user can declare an a-priori set of "possible premises". Each possible premise is a literal and every premise set in every query to the ATMS must be a subset of the set of possible premises.

In the ATMS algorithm, a *label* is a set of premise sets. Each proposition in the network has both a true and a false label. If proposition **P** has a true label consisting of premise sets $\Sigma_1, \ldots \Sigma_n$, then for each Σ_i we must have that **P** logically follows from Σ_i and the internal constraints. An analogous statement holds for the premise sets in the false label of **P**. The premise sets can be propagated through the constraints. For example, if Σ_1 is a premise set that is a member of the true label of **P**, Σ_2 is a premise set in the true label of **Q**, and $(\mathbf{P} \wedge \mathbf{Q}) \rightarrow \mathbf{R}$ is an internal constraint, then the premise set $\Sigma_1 \cup \Sigma_2$ can be added to the true label for **R**. The propagation process is initialized by inserting a singleton premise set into the label for each possible premise. For example, if **P** is a possible premise, then the singleton premise set $\{\mathbf{P}\}$ is inserted into the true label for **P**. After the universal propagation has been performed one can answer a query of the form **(follows-from? P** Σ**)** by checking to see if there exists a premise set on the true label of **P** that is a subset of Σ. An analogous test can be made for queries involving negative literals.

The total number of propagations, and the total number of premise sets generated, can be reduced by imposing two "filters" on the premise sets in labels. First, the premise sets must be "consistent". Most obviously, no premise set can contain both a proposition and its negation. In addition, however, one can use the special proposition **contradiction** to make a stronger consistency filter. No premise set in any label, other than the true label for **contradiction**, can contain as a subset any premise set that is a member of the label for **contradiction**. The second filter involves the notion of subsumption. If Σ_1 and Σ_2 are two premise sets in the same label, and Σ_1 is a proper subset of Σ_2, then the label Σ_2 can be removed. The ATMS runs the single universal propagation process to completion using these two filters to prune the premise sets in labels.

Universal Propagation and BCP. The notation $\Gamma, \Sigma \ \vdash_{\mathrm{BCP}} \ \Phi$ will be used to indicate that Φ can be derived from internal constraint set Γ and premise set Σ using Boolean constraint propagation. In the following discussion the symbol Γ will be used freely to denote a fixed but arbitrary internal constraint set. A *support* for a given proposition symbol **P** is a premise set Σ satisfying the following three conditions:

- $\Gamma, \Sigma \ \vdash_{\mathrm{BCP}}$ **P**
- $\Gamma, \Sigma \ \not\vdash_{\mathrm{BCP}}$ **contradiction**
- There is no proper subset Σ' of Σ such that $\Gamma, \Sigma' \ \vdash_{\mathrm{BCP}}$ **P**.

After the universal ATMS propagation procedure has run to completion, the premise sets in the true label of **P** are precisely the supports for **P**. This implies that the ATMS implements exactly the same behavior as the BCP-TMS.[3]

ATMS Complexity. The ATMS and BCP implementations generate exactly the same behavior at the level of the generic interface. Furthermore, the BCP implementation is guaranteed to require at most linear space and linear time per query. On the other hand, because a given label can contain an exponential number of different minimal premise sets, the ATMS implementation can require both exponential time and exponential space to answer a single query. This worst case behavior can be easily realized even in the case where the Boolean constraints are Horn clauses without implication cycles.

Why would anyone propose an algorithm that is exponential in both time and space as an improvement on an algorithm that is linear in both? The answer seems to be that the exponential ATMS algorithm has better performance in some applications. In cases where the label sets remain small query answering using pre-computed label sets is more efficient than context-switching in the BCP implementation. As problem size grows, however, the exponential cost of the universal ATMS propagation begins to dominate the potential savings at query time. Another reason for preferring the ATMS algorithm involves the additional functionality discussed in the next section.

Additional Functionality of the ATMS.
The ATMS universal propagation algorithm computes the *minimal* sets of assumptions necessary to derive a given formula. This feature is useful in device diagnosis where one wants to find the minimal number of possible faults that explains a given observed behavior [de Kleer and Williams, 1987] [de Kleer and Williams, 1989]. In fault diagnosis, however, one is often interested in premise sets that contain only a single fault.

1990b]. de Kleer's original ATMS only allowed Horn clause constraints and only allowed positive literals in queries. Except for these restrictions, the original ATMS is identical to the system described here.

[2] A similar universal propagation process for truth maintenance systems was developed independently by Drew McDermott [McDermott, 1983].

[3] It should be noted that the ATMS described here is different from the clause management system described in [de Kleer and Reiter, 1987]. The clause management system, or CMS, requires logical completeness. If constraints are restricted to Horn clauses and queries are restricted to positive literals, as in the original ATMS, BCP is logically complete and the CMS and BCP specifications are equivalent.

Under the single-fault assumption the full generality of the ATMS is not needed.

Applications to Search

It has often been said that truth maintenance systems are useful in controlling search [Doyle, 1979], [de Kleer, 1986a]. Because "search" is a loosely defined term, this claim is difficult to evaluate in general. Rather than attempt to evaluate the general claim, I will consider a more restricted class of search problems known as constraint satisfaction problems (CSPs).

A CSP consists of a set of variables, where each variable is associated with a finite set of possible values, plus a set of constraints. Each constraint consists of two of the given variables plus an enumeration of "allowed" pairs of values for those variables. An assignment of values to all the variables of a CSP is said to satisfy a given constraint if the pair of values assigned to the variables of the constraint is one of the allowed pairs of the constraint. A solution to a CSP is an assignment of values to the variables of the CSP that satisfies all of the constraints of the CSP. It is easy to show that determining the existence of a solutions to CSPs is NP-complete. This definition of a CSP can be generalized to allow constraints of more than two variables without changing the essentials of the analysis given below.

There is a large literature on algorithms for efficiently finding solutions to constraint satisfaction problems. The best introduction to the general theoretical framework is still perhaps Knuth's paper on methods of evaluating the running time of backtrack search [Knuth, 1975]. Knuth introduces the notion of a consistency test to be applied at each node of the backtrack search tree. Today there is a wide variety of possible consistency tests, the most effective of which are based on some form of constraint propagation [Mackworth, 1977], [Pearl and Korf, 1987]. In addition to constraint-propagation based consistency tests, there are a variety of heuristics for selecting which variable to instantiate next and which value for that variable to try first [Haralick and Elliot, 1980], [Freuder, 1985], [Dechter and Pearl, 1988], [Zabih, 1990]. Furthermore, there is variety of mechanisms for "backjumping", i.e., jumping back to earlier choice points because a "dependency analysis" shows that intervening choices where not involved in the cause of failure [Stallman and Sussman, 1977], [Gaschnig, 1979], [Bruynooghe and Pereira, 1984]. Research on algorithms for solving CSPs continues to be active.

Translating CSPs into Boolean Clauses. It is possible to use a TMS as the foundation of a procedure for solving arbitrary CSPs. Truth maintenance techniques operate on Boolean constraints rather than CSPs as defined above. To use a TMS in solving a CSP one can translate the CSP into a set of Boolean constraints. More specifically, for each variable X in C, with possible values $x_1, \ldots x_n$, we introduce the proposition symbols "$X = x_1$", ... "$X = x_n$" and the constraints

$$\text{“}X = x_1\text{”} \vee \text{“}X = x_2\text{”} \vee \cdots \vee \text{“}X = x_n\text{”}$$

and

$$\wedge_{i<j}(\neg\text{“}X = x_i\text{”} \vee \neg\text{“}X = x_j\text{”}).$$

These constraints form a set of disjunctive clauses that together are equivalent to the statement that exactly one of the propositions "$X = x_i$" is true. Now there are two simple ways of translating the constraints of the CSP into constraints on these proposition symbols. The first translation I will call the *negative translation.* Consider a constraint on variables X and Y. For each pair $<x, y>$ of possible values for X and Y respectively that *is not* an allowed pair of the constraint, we add the clause $\neg$"$X = x$" $\vee\neg$"$Y = y$". If C is a CSP (as defined above), I will let $N(C)$ be the set of Boolean clauses generated from the variables and constraints of C in this way. The second translation I will call the *positive translation.* Again consider a constraint on variables X and Y. For each possible value y of Y, let $x_1, \ldots x_k$ be the set of all possible values of X such that $<x_i, y>$ *is* an allowed pair of the constraint. For each possible value y of Y we add the constraint

$$\neg\text{“}X = x_1\text{”} \wedge \cdots \wedge \neg\text{“}X = x_k\text{”} \rightarrow \neg\text{“}Y = y\text{”}.$$

This implication is equivalent to a clause, i.e., a disjunction of literals. For any constraint satisfaction problem C, I will let $P(C)$ denote the positive translation of C into a set of disjunctive clauses. It is interesting to note that the size of the translation $N(C)$ is governed by the number of incompatible pairs of values in the constraints while the size of $P(C)$ is governed by the number of allowed pairs of values in the constraints.

Using a TMS to solve a CSP. Given a constraint satisfaction problem C it is possible to give either the Boolean constraint set $N(C)$ or the constraint set $P(C)$ to a TMS. A simple backtrack program can then be written to search the assignments of values to CSP variables where a partial assignment is encoded as a TMS premise sets of the form "$X_i = x_{i,j}$", ... "$X_k = x_{k,j}$". A filter can be imposed on the backtrack search by asking the TMS, at each node in the search tree, if the special node **contradiction** is derivable from the current partial assignment. If the positive translation has been used to convert the CSP to Boolean constraints (and the TMS simulates BCP) then this filter is equivalent to classical arc consistency.[4] If the negative translation has been used (and the TMS simulates BCP) then this filter is weaker, but more efficient than, classical

[4]The linear time of the BCP algorithm on clausal constraints implies that arc consistency can be achieved in time proportional to the number of consistent pairs of values for constrained pairs of variables. Thus BCP provides an alternative algorithm for achieving the arc-consistency complexity bound given in [Mohr and Henderson, 1986].

arc consistency. The filter that results from BCP applied to the negative translation might be called BCP consistency.

In addition to providing powerful search filters, the justification facility of a TMS provides a mechanism for performing a certain form of backjumping known as dependency directed backtracking [Stallman and Sussman, 1977]. When a failure occurs in the search process one can use the justification mechanisms to uncover the subset of the current premise set that was used to derive the special proposition **contradiction**. This allows a new constraint to be installed called a "nogood" which states that at least one of these premises must be false. This new derived constraint allows BCP to make more inferences than were possible with the old constraints and the consistency filter on the remaining search becomes stronger.

Dependency directed backtracking, and backjumping in general, are particularly useful when, for some reason, a poor choice has been made in selecting the order of the variables to be instantiated in the backtrack search process. To my knowledge, no one has established the pragmatic value of backjumping in a search that already does careful variable ordering and constraint propagation based consistency testing. Because of the potential for generating an exponential number of additional derived constraints, dependency directed backtracking is a particularly expensive form of backjumping.

The ATMS universal propagation algorithm can also be applied to the Boolean translation of a CSP [de Kleer, 1989]. In addition to clauses generated by the translation, one can specify each assumption of the form "$X = x$" as a possible premise. An additional proposition called **all-variables-assigned** can be introduced such that BCP can derive the proposition **all-variables-assigned** if and only if a proposition of the form "$X = x$" has been derived for each variable X. After running the ATMS universal propagation procedure the set of all solutions to the CSP is contained in the true label of the proposition **all-variables-assigned**. Note that all variable assignments that violate the given constraints are automatically removed by the consistency filtration of premise sets in the universal ATMS algorithm. In this way, the universal ATMS propagation procedure yields "choices without backtracking" [de Kleer, 1984].

The most efficient and natural justification structure for the proposition **all-variables-assigned** will cause the ATMS universal propagation procedure to simulate a backtrack search with a fixed order in which the variables are considered. However, the backtrack search will be done in space-intensive breadth-first manner rather than a space-efficient depth-first manner. In general, space-intensive breadth-first searches are considered to be less efficient (in both time and space) than space-efficient depth-first searches [Pearl and Korf, 1987]. To my knowledge, the ATMS has never been shown to be more time efficient for enumerating solutions to a CSP than classical backtracking approaches.

Truth maintenance systems may be useful as a general method of solving CSPs because they provide a general and efficient mechanism for constraint propagation based consistency testing of partial assignments. The other ways in which TMS technology might be applied to CSPs, i.e., dependency directed backtracking and ATMS universal propagation, appear to be of limited value. The ATMS universal propagation procedure is probably more appropriate for finding *minimal* premise sets satisfying some condition.

General Problem Solving

Truth maintenance systems have often been viewed as an integral part of "AI languages", i.e., knowledge representation and programming languages that are designed to allow for the rapid development of expert systems or general problem solvers [de Kleer, 1986c], [de Kleer, 1986b]. A recent, and highly successful, incorporation of the constraint propagation aspects of truth maintenance into a general purpose language is Van Hentenryck's version of Prolog called Chip (for constraint handling in Prolog) [Van Hentenryck, 1989]. Van Hentenryck's version of Prolog is only tenuously related to the literature on truth maintenance systems. However, it seems that languages that combine automatic backtracking with automatic constraint propagation will be a major competitor with TMS-based AI languages in the future.

I will use the term "Van Hentenryck language" as a general term for any programming language that combines automatic backtracking with automatic constraint propagation. Rather than describe Van Hentenryck's version of Prolog, I will describe a Van Hentenryck dialect of Lisp called Chil (for constraint handling in Lisp). Actually, Chil is built on Scheme, a dialect of Lisp that provides **call-with-current-continuation**. Automatic backtracking can be incorporated into Scheme by adding two new primitives: **either** and **fail**. The special form **either** takes two arguments and non-deterministically returns the value of one of them.[5] The procedure **fail** causes the computation to be restarted from the most recent non-deterministic choice. Given Scheme's implementation of **call-with-current-continuation**, the "primitives" **either** and **fail** can be efficiently implemented in about ten additional lines of code. Other standard Prolog primitives, such as **cut** and **bag-of** are implemented with similar ease.

The procedures **either** and **fail** provide automatic backtracking and allow the concise expression of a large variety of backtrack search programs. Automatic con-

[5] **either** is a version of McCarthy's **amb** [McCarthy, 1963]. The word **either** reads more naturally.

straint propagation is added with three more procedures: **make-domain-object**, **add-constraint**, and **force-value**. The procedure **make-domain-object** takes one argument which is a list of "possible values" and returns a data structure that represents a CSP variable with the given set of possible values. The procedure **add-constraint** takes three arguments: two domain objects (objects returned by **make-domain-object**) and an ordinary Scheme predicate of two arguments. The procedure **add-constraint** installs a constraint stating that the simultaneous values of the two domain objects must satisfy the given predicate. Finally, the procedure **force-value** takes a domain object and non-deterministically assigns that domain object one of its possible values. The procedure **force-value** automatically invokes constraint propagation. If constraint propagation does not result in a constraint violation then **force-value** returns the selected value. Otherwise backtracking occurs. Using these primitives, a constraint propagation version of the n-queens problem can be expressed as follows.

```
(define (n-queens n)
  (let ((queen-variables
         (map (lambda (ignore)
                (make-domain
                  (numbers-between 1 n)))
              (numbers-between 1 n))))
    (do-from-to (i 1 (- n 1))
      (do-from-to (j (+ i 1) n)
        (add-constraint
         (nth i queen-variables)
         (nth j queen-variables)
         (lambda (v1 v2)
           (and (not (= v2 v1))
                (not (= v2 (+ v1 (- j i))))
                (not (= v2 (- v1 (- j i)))))))))
    (map force-value queen-variables)))
```

The above program uses **numbers-between**, **do-from-to**, and **nth** which are not Scheme primitives, but which are easily defined. Given the above procedure, the expression **(bag-of (n-queens 8))** evaluates to a list of all 92 solutions of the 8-queens problem. Furthermore, the search process specified by the above procedure uses constraint propagation based consistency testing at each node of the search tree. Van Hentenryck has shown that the addition of automatic constraint propagation to languages with automatic backtracking can drastically improve the performance of a wide variety of useful backtrack search programs.

Strengthening Constraint Propagation

Constraint propagation appears to be of central importance in search-based problem solving. One way of attempting to discover more powerful constraint propagation techniques is to study the relationship between constraint propagation and inference. Constraint propagation is a form of inference — values for unassigned variables are deduced from the values already assigned. The relationship between inference and constraint propagation can be made explicit by characterizing constraint propagation processes in terms of inference rules. Boolean constraint propagation can be defined in terms of a certain (incomplete) set of inference rules for Boolean logic [McAllester, 1989]. Van Hentenryck also defines the various constraint propagation techniques used in Chip in terms of rules of inference. In fact, virtually any form of constraint propagation can be defined in terms of rules of inference.

Constraint propagation inference rules are unusual, as rules of inference, in that it is possible to determine in polynomial time whether a given statement can be derived from given premises. In other words, constraint propagation inference rules generate a polynomial time decidable inference relation — such relations will be called *tractable*. Although every constraint propagation technique corresponds to a tractable inference relation, there are many tractable inference relations that do not correspond to any standard constraint propagation technique. For example, the inference rules that define BCP can be combined with the standard inference rules for equality, including the substitution of equals for equals, and the resulting rule set is still polynomial time decidable [McAllester, 1989]. In [McAllester *et al.*, 1989] it is argued that the power of tractable rule sets for first order inference is sensitive to the syntax in which formulas are expressed — an alternative syntax based on taxonomic relationships between classes yields a more powerful tractable rule set. In [McAllester and Givan, 1989] it is argued that the power of the tractable rule set can be further improved if the syntax is based on the specifier structure of natural language noun phrases under Montague semantics. This observation provides a functional justification for some of the syntactic features of natural language. In [McAllester, 1990] a general theory of tractable rule sets is presented and an algorithm is given for automatically recognizing tractability in rule sets.

Inference is closely related to constraint propagation and constraint propagation is clearly important in improving search efficiency. People seem to use inference to drastically reduce the amount of search required in problem solving. It seems possible that some of the power of human inference is based on generalizations of constraint propagation — powerful yet tractable inference relations computed in a fraction of a second. Perhaps there is still hope for the construction of efficient general purpose inference mechanisms.

References

[Bruynooghe and Pereira, 1984] M. Bruynooghe and L. M. Pereira. Deduction revision by intelligent backtracking. In J. A. Cambell, editor, *Implementations of Prolog*, pages 196–215. Ellis Horwood, 1984.

[Dechter and Pearl, 1988] R. Dechter and J. Pearl. Network-based heuristics for constraint-satisfaction problems. *Artificial Intelligence*, 34:1–38, 1988.

[de Kleer and Williams, 1987] J. de Kleer and B. Williams. Diagnosing multiple faults. *Artificial Intelligence*, 32:97–130, 1987.

[de Kleer and Williams, 1989] J. de Kleer and B. Williams. Diagnosis with behavioral modes. In *Proceedings IJCAI-89*, pages 104–109, 1989.

[de Kleer and Reiter, 1987] J. de Kleer and R. Reiter. Foundations of assumption-based truth maintenance systems. In *AAAI87*, pages 183–188, 1987.

[de Kleer, 1984] J. de Kleer. Choices without backtracking. In *Proceedings of AAAI-84*, pages 79–85, 1984.

[de Kleer, 1986a] J. de Kleer. An assumption-based tms. *Artificial Intelligence*, 28:127–162, 1986.

[de Kleer, 1986b] J. de Kleer. Extending the atms. *Artificial Intelligence*, 28:163–196, 1986.

[de Kleer, 1986c] J. de Kleer. Problem solving with the atms. *Artificial Intelligence*, 28:197–224, 1986.

[de Kleer, 1989] de Kleer, J., A comparison of ATMS and CSP techniques, *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, Detroit, MI (August 1989).

[de Kleer, 1990a] de Kleer, J., Exploiting locality in the ATMS, *AAAI-90*, Boston, Mass. (August 1990).

[de Kleer, 1990b] J. de Kleer. A practical clause management system. SSL Paper P88-00140, Xerox PARC, submitted for publication.

[Doyle, 1979] J. Doyle. A truth maintenance system. *Artificial Intelligence*, 12:231–272, 1979.

[Freuder, 1985] E. C. Freuder. A Sufficient Condition for Backtrack-Bounded Search. *J. ACM*, 32(4):755-761, 1985.

[Gaschnig, 1979] J. Gaschnig. Performance measurement and analysis of certain search algorithms. Report, CMU, 1979.

[Gelfond and Lifschitz, 1988] M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In *Logic Programming: Proceedings of the Fifth International Conference and Symposium*, pages 1070–1080, 1988.

[Gelfond, 1989] M. Gelfond. Autoepistemic logic and the formalization of common sense reasoning: Preliminary report. In *Non-Monotonic Reasoning: 2nd International Workshop (Lecture Notes in Artificial Intelligence 346)*, pages 176–189. Springer-Verlag, 1989.

[Haralick and Elliot, 1980] R. Haralick and G. Elliot. Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence*, 14:263–313, 1980.

[Van Hentenryck, 1989] P. Van Hentenryck. *Constraint Satisfaction in Logic Programming.* MIT Press, 1989.

[Knuth, 1975] D. Knuth. Estimating the efficiency of backtrack programs. *Mathematics of Computation*, 29(129):121–136, January 1975.

[Konolige, 1987] K. Konolige. On the relationship between default theories and non-monotonic logic. *Artificial Intelligence*, 35:343–382, 1987.

[Mackworth, 1977] A. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8(1):99–181, 1977.

[McAllester and Givan, 1989] D. McAllester and R. Givan. Natural language syntax and first order inference. Memo 1176, MIT Artificial Intelligence Laboratory, October 1989.

[McAllester *et al.*, 1989] D. McAllester, R. Givan, and T. Fatima. Taxonomic syntax for first order inference. In *Proceedings of the First International Conference on Principles of Knowledge Representation and Reasoning*, pages 289–300, 1989.

[McAllester, 1980] D. McAllester. An outlook on truth maintenance. Memo 551, MIT Artificial Intelligence Laboratory, August 1980.

[McAllester, 1989] D. McAllester. *Ontic: A Knowledge Representation System for Mathematics.* MIT Press, 1989.

[McAllester, 1990] D. McAllester. Automatic recognition of tractability in inference relations. Memo 1215, MIT Artificial Intelligence Laboratory, February 1990.

[McCarthy, 1963] J. McCarthy. A basis for a mathematical theory of computation. In P. Braffort and D. Hirschberg, editors, *Computer Programming and Formal Systems.* North-Holland, 1963.

[McCarthy, 1986] J. McCarthy. Applications of circumscription to formalizing common sense reasoning. *Artificial Intelligence*, 26:89–118, 1986.

[McDermott, 1983] D. McDermott. Contexts and data dependencies: a synthesis. *IEEE Transaction on Pattern Analysis and Machine Intelligence*, 5(3):237–246, 1983.

[Mohr and Henderson, 1986] R. Mohr and T. Henderson. Arc and path consistency revisited. *Artificial Intelligence*, 28(2):225–233, 1986.

[Pearl and Korf, 1987] J. Pearl and R. Korf. Search techniques. *Ann. Rev. Comut. Sci.*, 2:451–467, 1987.

[Pearl, 1988] J. Pearl. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference.* Morgan Kaufmann, 1988.

[Stallman and Sussman, 1977] R. Stallman and G. Sussman. Forward reasoning and dependency directed backtracking in a system for computer-aided circuit analysis. *Artificial Intelligence*, 9:135–196, 1977.

[Touretzky, 1986] D. Touretzky. *The Mathematics of Inheritance Systems.* Morgan Kaufmann, 1986.

[Zabih, 1990] R. Zabih, Some applications of graph bandwidth to constraint satisfaction problems. *AAAI-90.*