Computer Science Department Technical Report
University of California
Los Angeles, CA 90024-1596

STRUCTURE-DRIVEN ALGORITHMS FOR TRUTH MAINTENANCE

Rina Dechter
Avi Dechter

July 1991
CSD-910045

Technical Report
R-138
February 1990

STRUCTURE-DRIVEN ALGORITHMS FOR TRUTH MAINTENANCE

Rina Dechter

Computer Science Department,
Technion - Israel Institute of Technology

Avi Dechter

School of Business administration,
Cal-State Northridge,

ABSTRACT

This paper presents distributed algorithms for performing truth-maintenance tasks on singly-connected structures. We show that on this model the JTMS's belief maintenance and consistency maintenance tasks are linear in the network size while the ATMS task remains exponential with a reduced exponent -- the branching degree of the network. Although the model is restricted, it serves three purposes. First, it helps in identifying the source of the computational difficulties associated with both JTMS and ATMS. Second, efficient algorithms on singly connected models may be adapted to general structures by known clustering techniques. Finally, these algorithms can serve as approximation or as heuristics for processing general truth maintenance problems.

1. Introduction

Reasoning about dynamic environments is a central issue in Artificial Intelligence. When dealing with a complex environment, only partial description of the world is known explicitly at any given time. A complete picture of the environment can only be speculated by making assumptions, which must maintain a consistent view of the world. When new facts become known, it is important to maintain the consistency of our view of the world so that queries of interest (e.g., is a certain proposition currently believed to be true?) can be answered coherently at all times.

Truth maintenance systems (TMSs) are computational schemes which are intended to handle such situations. TMSs reason non-monotonically, namely, they allow inference systems to draw conclusions and retract them in the context of incomplete and changing information. In its generic form, a TMS is concerned with propositions (representing potential beliefs) which it attempts to prove (thus turning them into beliefs) by using a set of inference rules which constitute its knowledge base. In addition, most TMSs recognize two special types of propositions which do not require a proof: premises and assumptions. A premise is a proposition which represents an observed fact or an a' priori believed fact. An Assumption is a proposition which is presumed to be true under normal conditions but may be retracted if it contradicts with the observed facts or with other assumptions.

Two main approaches to TMS design have emerged: the JTMS (Justification-based TMS) [10, 16] and the ATMS (Assumption-based TMS) [8]. A JTMS undertakes to determine provable propositions and provide such proofs (called justifications) described by a mixture of known facts, and a **given** set of assumptions. In ATMS the system maintains for each proposition a whole collection of assumption sets (called environments) any of which can be used to prove the proposition given the facts.

An important part of a TMS's functions is that of constraint satisfaction. Several authors have discussed the connections between truth maintenance systems and constraint satisfaction problems and the potential benefits to both of better understanding these relationships, e.g. [9], and [18]. The main thrust of these efforts has been to show search reduction techniques developed in one area may be used to the benefit of the others.

In this paper we pose the basic JTMS and ATMS tasks using the language of constraints and propose algorithms for their achievement. Our algorithms rely on special techniques that were developed in recent years for solving "pure" constraint satisfaction problems. The main characteristic of these techniques are that they are ''sensitive'' to the structure of the problem so as to take advantage of special structures. The use of the constraint framework for examining these tasks and for specifying the algorithms has the additional benefit of facilitating the analysis of the complexity of these tasks.

As a departure point for the ''leap'' from TMS to CSP we use the concept of a clause management system (CMS) proposed by Reiter and de_Kleer [19] to describe the tasks of truth maintenance systems (ATMS in particular) in propositional logic. The language of constraints has the same expressive power as propositional logic but expresses more naturally the idea that certain sets of propositions are mutually exclusive and exhaustive, a notion which is implicit in the concept of **variables** and **relations**.

The remainder of this paper is organized as follows: Section 2 summarize the Clause management system and defines the ATMS and JTMS's task's, and section 3 transforms the CMS formulation into constraint networks' language. Sections 4,5,6 and 7 focus on algorithms for performing the JTMS's task on singly connected networks, while sections 8, 9, 10, and 11 focus on algorithms for the ATMS task on singly connected networks. Section 12 provides a summary and concluding remarks.

2. JTMS and ATMS Tasks

Following Reiter and de-Kleer [19] we choose to express the tasks of a JTMS and an ATMS in propositional logic using their concept of a Clause Management System (CMS).

The CMS assumes a propositional language with countably infinitely many propositional symbols and the logical connectives $\cup$ and $\neg$. The formulas of the language are defined in the usual way. Let S be a set of formulas and w a formula. Then S is said to entail w, denoted $S \models w$ just in case every assignment of truth values to the propositional symbols of the language which makes each formula of S true also makes w true.

A literal is a propositional symbol or the negation of a propositional symbol. A clause is a finite disjunction of literals, with no literal repeated. Let Σ be a set of clauses and C a clause. A clause S is a support for C with respect to Σ iff $\Sigma \not\models S$ and $\Sigma \models S \cup C$.

A support clause S for C with respect to Σ has the following properties:

1. Σ *entail* $S \cup C$, i.e., Σ *entail* $notS \supset C$.

2. Σ *does not entail* S, i.e., $\Sigma \cup \{notS\}$ is satisfiable.

The first property means that the conjunction of literal *notS* is an hypothesis which, if added to Σ would make conclusion C true. The second property means that *notS* is consistent with Σ.

The notion of support may now be used to specify two fundamental tasks which a CMS may be expected to achieve depending on the type of query it receives from the reasoner. On the one hand the reasoner may query the CMS with a clause S representing an hypothesis *notS* which is consistent with Σ. The task of the CMS is to find all (or some) clauses C which are supported by S with respect to Σ. We refer to this as the fundamental JTMS task. On the other hand, the reasoner may query the CMS with a clause C, representing set of potential beliefs. The task of the CMS in this case is to determine all minimal support clauses for C with respect to Σ.

This is the fundamental ATMS task.

Since the definition of S being a support to C with respect to Σ is dependent on S being consistent with Σ, there is another task that the CMS must be able to achieve, namely, given a clause S, determine whether it is consistent with Σ, and if not, suggest a minimal subset of literals of Σ such that if removed will make the set of clauses consistent with S.

The notion of assumptions is fundamental for systems that are trying to reason non-monotonically. To equip the CMS with such capabilities a distinct set of propositional symbols, $\{A_1, \ldots, A_n\}$ called **assumption symbols**, is introduced, where a particular truth assignment to the assumption symbols is referred to as **assumptions**. We now formally define the JTMS and ATMS tasks within this model:

1. **Answer the JTMS query** : given a knowledge base J consisting of a set of clauses together with a current truth assignment to the assumption symbols, and given a query α, determine whether or not α is entailed by J.

2. **Consistency maintenance in JTMS:** given a consistent knowledge-base, J, consisting of a set of clauses together with a current truth assignment to the assumption symbols, and given an assertion α determine if $J \cup \alpha$ is consistent and if not, find a minimal subset of assumptions whose truth assignment should be reversed to restore consistency.

3. **Answer the ATMS query** : given a knowledge-base, J, consisting of propositional clauses and assumption symbols, and given a literal α, compute all minimal supports to α restricted to assumption symbols. In other words, compute all sets of consistent assumptions $\{A_1, \ldots, A_r\}$ such that $\{\neg A_1 \mid, \ldots, \mid \neg A_r\}$ constitute a minimal support to α w.r.t. J [8]

Since in ATMS framework no commitment to assumptions is being made, determining the consistency of a set of clauses is part of the task of finding consistent supports. However, an inconsistency of the knowledge when no assumptions are committed implies an inherent inconsistency which is not curable by changing assumptions.

Reiter et. al [19]. discuss two approaches to CMS, **interpreted and compiled.** In the interpreted approach clauses submitted to the reasoner are stored as they are (maybe indexed by their literal), and when a query arrives, query-processing algorithms produce the answers. In the compiled approach, all answers to all possible queries are computed and saved ahead of time. Thus, in the former mode the costly operations are query processing and consistency maintenance, while in the latter mode query processing takes constant time but there is a space-overhead and an extra work for maintaining the consistency of the compiled data. Traditional TMSs were dominated by the compiled approach presumably assuming that maintenance overhead is compensated by a constant query answering.

3. Constraint-formulation for CMS

A **constraint satisfaction problem** (CSP), also referred to as a **constraint-network** (CN), involves a set of n variables $X_1,...,X_n$, each represented by its domain values, $R_1, \dots, R_n$, and a set of constraints. A constraint $C_i(X_{i_1}, \cdots, X_{i_j})$ is a subset of the Cartesian product $R_{i_1} \times \cdots \times R_{i_j}$ that specifies which values of the variables are compatible with each other. A solution (also called an extension) is an assignment of values to all the variables which satisfy all the constraints, and the most common task associated with these problems is to find one or all solutions. A constraint network is **inconsistent** if it has no solutions. A constraint is usually represented by the set of all tuples which are not forbidden by it. A **binary CSP** is one in which all the constraints are binary, i.e., they involve only pairs of variables. A binary CSP can be associated with a **constraint-graph** in which nodes represent variables and arcs connect pairs of variables which are constrained explicitly. Figure 1 presents a Constraint-Network (modified from [14]). Each node represents a variable whose values are explicitly indicated, and each link is labeled with the set of value-pairs permitted by the constraint between the variables it connects (observe that the constraint between connected variables is a strict lexicographic order along the arrows.)

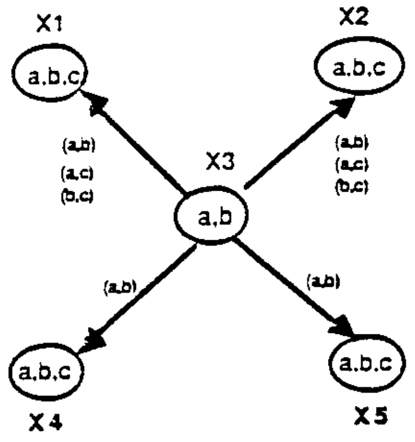
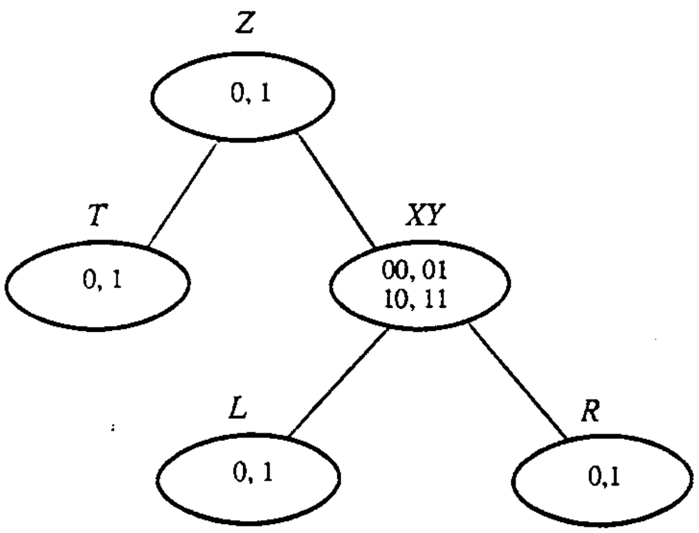


Figure 1: An example of a binary CN

A CMS can be viewed as a constraint network. Each propositional symbol can be regarded as a bi-valued variable having truth values "1" (truth) and "0" (false). A clause defines a constraint on the corresponding subset of variables given by its truth table and a set of clauses is a constraint-network. For instance the clause $T \rightarrow Z$ can be expressed as a relation on the pair of bi-valued variable T and Z as: $TZ = \{11,01,00\}$. Consider the following set of propositions: $\{X \mid Y \rightarrow Z,\ T \rightarrow Z,\ L \rightarrow X,\ R \rightarrow Y\}$ which can be modeled as a CSP with the four bi-valued variables T,Z,R,L and the compound variable XY having the domain values, 00, 01,10,11 (each pair is a name of one value). The explicit form of the constraints is generated via the truth table of implication. The constraint graph and the explicit constraints are depicted in figure 2.

We now extend the notion of entailment to constraint network terminology. We say that a constraint C is **entailed** (or believed or holds) by a network of constraints, R, if it is satisfied by **every solution of** R. Following that, the notion of **support** for a constraint C w.r.t. network R is



$T \rightarrow Z$

T	Z
1	1
0	1
0	0

$R \rightarrow Y$

R	XY
1	01
1	11
0	01
0	11
0	00
0	10

$L \rightarrow X$

L	XY
1	10
1	11
0	10
0	11
0	00
0	01

$X \mid Y \rightarrow Z$

XY	Z
01	1
10	1
11	1
00	1
00	0

Figure 2: An example CSP

defined as a consistent instantiation of a subset of the variables, $(X_1 = x_1, \ldots, X_k = x_k)$ such that C holds in the restriction of R to this subset-instantiation, namely, in any solution to the network that satisfies the partial instantiation, C holds. (the support here is the negated version of the support in CMS). The support is minimal if no subset of the instantiation is a support to C w.r.t. R. To model assumptions within constraint network we allow the existence of a distinct set of variables, called **assumption variables**, whose possible values are called **assumptions**, thus assumption symbols are naturally mapped to assumption variables.

Assumption variables can be used to model default rules: e.g. birds fly (unless they are dead), an adder functions correctly (unless it is faulty). This is accomplished by adding to the constraint representing the rule an assumption variable with two values: one representing the default assumption (e.g., that the rule "birds fly" is true) and the other representing the exception.

The JTMS task rephrased into the CN model is to **determine whether or not an assignment, $X = x$, holds in all solutions** given some instantiations of the assumption variables. The task of ATMS is to find all minimal instantiations of assumption variables which support a query

C, given a network R. Therefore, if the query is $X = x$, the task is to find a minimal instantiation of assumptions which make the network consistent with the value x of X and at the same time **not consistent with any other value of X.**

In this paper we present algorithms for performing JTMS and ATMS tasks. We first assume a restricted problem structure of singly connected networks with binary constraints, then extend it to non-binary singly connected topologies. Although it is a restricted case it serves three purposes. Being a simplified model, it helps identify the source of the computational difficulties associated with both JTMS and ATMS. Second, efficient algorithms on this model may be adapted to general networks using a tree-clustering transformation [6] or the cycle-cutset method [4] Thirdly, these algorithms can also serve as approximation algorithms or as heuristics for general networks.

4. Determining belief via support propagation

In this section we present a distributed algorithm for determining the status of each proposition of the form $X=x$, namely, determining whether or not it holds in all solutions. To accomplish this task we compute for each value of a variable the number of solutions in which it participates. We call this figure the support **number** to distinguish it from the notion of support sets as defined earlier. The support numbers can be thought of as measuring the degree of belief in the proposition represented by the value. (If the set of all solutions was assigned a uniform probability distribution, then the of support number is precisely the marginal probability of the proposition, namely, its "belief" in the corresponding Bayes network [17]). In particular a proposition $X=x$ is entailed if it has a positive support and at the same time the support for other values X's domain is 0. The support figures for all values in a variable's domain constitute a **support vector** for that variable.

It is well known that constraint networks whose constraint graph is a tree can be solved easily [15, 11, 3]. Consequently, the number of solutions in which each value in the domain of each variable participates (namely, the support number of this value), can also be computed very efficiently. In this section we present a distributed scheme for calculating the support vectors for all variables, and for their updating to reflect changes in the network.

Consider a fragment of a tree-network as depicted in Figure 3.

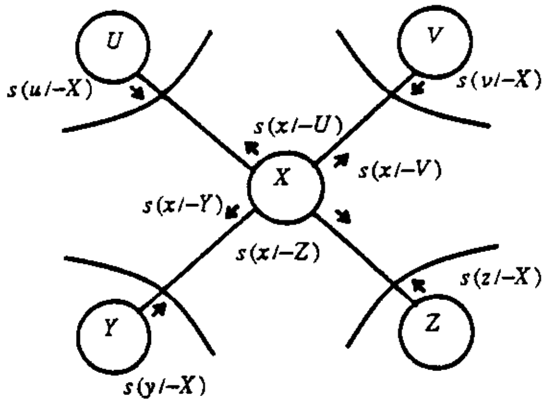


Figure 3: A fragment of a tree-structured CN

The link (X,Y) partitions the tree into two subtrees: the subtree containing X, $T_{XY}(X)$, and the subtree containing Y, $T_{XY}(Y)$. Likewise, the links (X,U), (X,V), and (X,Z), respectively, define the subtrees $T_{XU}(U)$, $T_{XV}(V)$ and $T_{XZ}(Z)$. Denote by $s_X(x)$ the overall support number for value x of X, by $s_X(x/Y)$ the partial support for $X = x$ **contributed by subtree $T_{XY}(Y)$** (i.e., the number of solutions of this subtree which are consistent with $X = x$), and by s_X(x/-Y) the support for $X = x$ in $T_{XY}(X)$. (These notations will be shortened to $s(x)$, $s(x/Y)$ and $s(x/-Y)$, respectively, whenever the identity of the variable is clear.) The support for any value x of X obeys:

$$s(x) = \prod_{Y \in X\text{'s neighbors}} s(x/Y)\,, \tag{1}$$

namely, it is a product of the supports contributed by each neighboring subtree. The support that Y contributes to $X = x$ can be further decomposed as follows:

$$s(x/Y) = \sum_{(x,y) \in C(X,Y)} s(y/-X)\,, \tag{2}$$

when $C(X,Y)$ denotes the constraint between X and Y. Namely, since x can be associated with several **matching** values of Y, its support is the sum of the supports of these values. Equalities (1) and (2) yield:

$$s(x) = \prod_{Y \in X\text{'s neighbors}} \sum_{(x,y) \in C(X,Y)} s(y/-X) . \tag{3}$$

Equation (3) lends itself to the promised propagation scheme. If variable X gets from each neighboring node, Y, a vector of restricted supports, (referred to as **the support vector from Y to X**):

$$(s(y_1/-X), \ldots, s(y_t/-X)) , \tag{4}$$

where y_i is in Y's domain, it can calculates its own support vector according to equation (3) and, at the same time, generate an appropriate message to each of its own neighbors. The message X sends to Y, $(s(x/-Y))$, is the support vector reflecting the subtree $T_{XY}(X)$, and can be computed by:

$$\text{for every } \; x \in X \;\; s(x/-Y) = \prod_{Z \in X\text{'s neighbors},\, Z \neq Y} \sum_{(x,z) \in C(X,Z)} s(z/-X) . \tag{5}$$

The message generated by a leaf-variable is a vector consisting of zeros and ones representing, respectively, legal and illegal values of this variable.

In an **interpreted mode** the knowledge consists of the constraint network only. Then, when a request to determine the status of each value (i.e., whether or not, for every $x \in X$, $X = x$ is entailed) arrives, the following algorithm will generate all the support vectors. The computation consists of nodes sending to their neighbors the partial support vectors (4) whenever they are readily computed. When all nodes have received all the partial supports the overall supports can be computed using (3). The basic algorithm for node, X, having neighbors $Y_1, \ldots, Y_r$ is as follows:

```
propagate-partial supports ($X, Y_1, \dots, Y_r$)
1. begin
2.  for each neighbor $Y_j$ do
3.   if received partial supports from $Y_1, \dots, Y_{j-1}, Y_{j+1}, \dots, Y_r$ then
4.      compute $(s(x_1/-Y_j), \dots, s(x_k/-Y_j))$ using equation (5)
      and sent it to $Y_j$.
5.   if received partial supports from all neighbors then
6.       compute your overall support using (3).
7. end.
```

Clearly, when the algorithm terminates all nodes will have the partial and overall supports correctly computed. The message complexity and the time complexity is exactly $2e$ where e is the number of edges, and it equals $2n$ for trees. If only the support status of variable, X, is required the computation can be further simplified. First, a directed tree rooted at X should be generated, and then the partial support messages will be propagated in one direction only; from child nodes to their parents. At termination only X's supports are computed but the amount of message passing and the time complexity reduces from $2n$ to n (one message per link).

In a **compiled mode** we assume that the network computed and saved all the partial support vectors and the task is to update these vectors when a new input arrives. The updating scheme is initiated by a variable directly exposed to the new input. Such variable will recalculate and deliver the partial support vectors to each of its neighbors. When a variable in the network receives an update-message from a neighbor, it recalculates its outgoing messages, (i.e., messages to all other neighbors), and at the same time updates its own support vector. The propagation generated due to a single outside change will spread through the network only once (no feed-back), since the network has no loops.

To illustrate the mechanics of the propagation scheme in the compiled mode, consider again the problem of Figure 1. In Figure 4a the support vectors and the different messages are presented. The order within a support vector corresponds to the order of values in the originating variable, e.g., message (8,1) from X_3 to X_1 represents $(s_{X_3}(a/-X_1)\,,\, s_{X_3}(b/-X_1))$. Suppose now that the system is forced by an outside change to restrict the value of X_2 to "b". In that case X_2 will originate a new message to X_3 of the form (0,1,0). This, in turn, will cause X_3 to update its supports and generate updated messages to X_1,X_4 and X_5 respectively. The new supports and the new updated messages are illustrated in Figure 4(b).

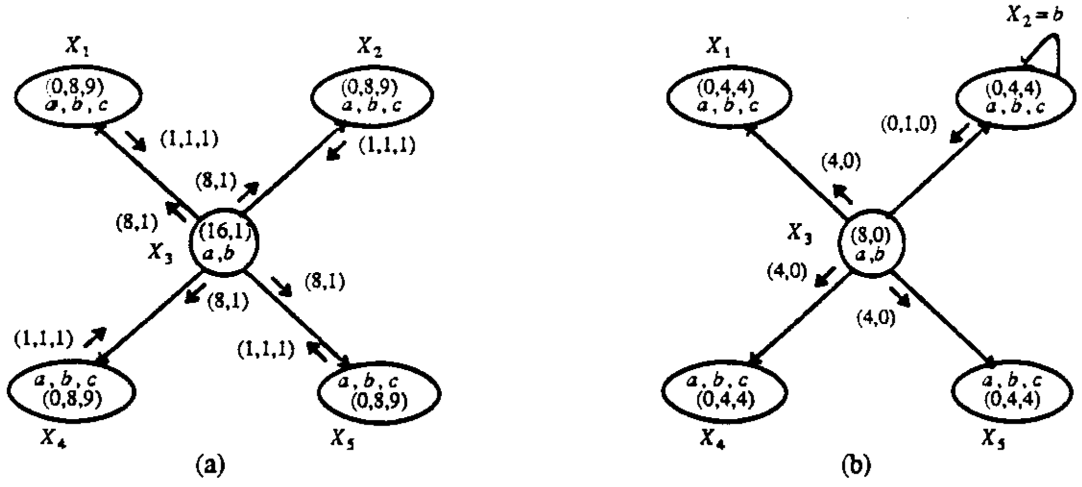


Figure 4: Support vectors before and after a change

Query processing takes constant time in the compiled version at the expense of a linear space overhead of $2n$. The message and time complexity for each update is exactly n.

If one is not interested in calculating numerical supports, but merely in indicating whether a given value has **some** support (i.e., participates in at least one solution), then **flat** support-vectors, consisting of zeros and ones, can be propagated in exactly the same way, except that the summation operation in (3) should be replaced by the logic operator **OR**, and the multiplication can be replaced by **AND**.

5. Handling Assumptions and Contradictions

When, as a result of new input, the network enters a contradictory state, (i.e., no solution exists) it often means that the new input is inconsistent with the **current set of assumptions**, and that some of these assumptions must be modified in order to restore consistency. The task of restoring consistency by changing the values assigned to a subset of the assumption variables is called **contradiction resolution.**

The subset of assumption variables that are modified in a contradiction resolution process should be minimal, namely, it must not contain any proper subset of variables whose simultaneous modification is sufficient for that purpose. A sufficient (but not necessary) condition for this set to be minimal is for it to be as small as possible. In this section we show how to identify a minimum number of assumptions that need to be changed in order to restore consistency. Unlike the support propagation scheme, however, the contradiction resolution process has to be synchronized. Assume that a variable which detects a contradiction propagates this fact to the entire network, creating in the process a directed tree rooted at itself. Given this tree, the contradiction resolution process proceeds as follows.

With each value v of each variable V we associate a weight $w(v)$, indicating the minimum number of assumption variables that must be changed in the directed subtree rooted at V in order to make v consistent in this subtree. These weights obey the following recursion:

$$w(v) = \sum_{Y_i} \min_{(v,y_{ij}) \in C(V,Y_i)} (\delta_{y_{ij}} + w(y_{ij})) , \tag{6}$$

where $\{Y_i\}$ are the set of V's children and their domain values are indicated by y_{ij}; i.e. y_{ij} is the j^{th} value of variable Y_i, and $\delta_{y_{ij}} = 1$ if y_{ij} is not a currently selected assumption value or "0" otherwise (see Figure 5). The weights of leaf values are "0" unless they represent not currently selected values, in which case they are "1". The computation of the weights is performed distributedly and synchronously from the leaves of the directed tree to the root. A variable waits to get the weights of all its children, computes its own weights according to (6), and sends them to its parent. During this **bottom-up-propagation** a pointer is kept from each value of V to the values in each of its child-variables, where a minimum is achieved. When the root variable X receives all the weights, it computes its own weights and selects one of its values that has a minimal weight. It then initiates (with this value) a **top-down propagation** down the tree, fol-

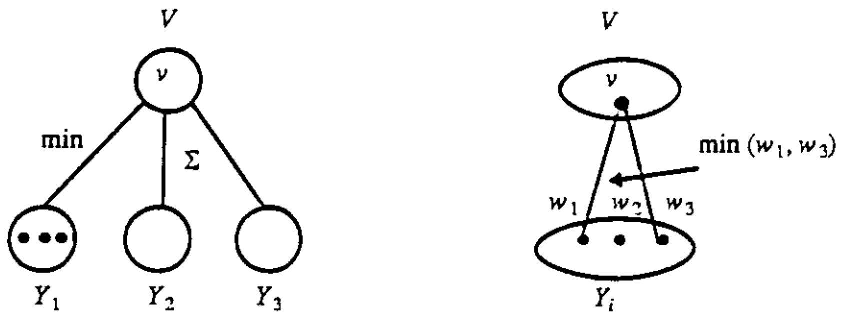


Figure 5: Weight calculation for node v

lowing the pointers marked in the bottom-up-propagation, a process which generates a consistent selection of assumptions with a minimum number of assumptions changed. At termination this process marks the assumption variables that need to be changed and the appropriate changes required.

There is no need, however, to activate the whole network for contradiction resolution, because the support information clearly points to those subtrees where no assumption change is necessary. Any subtree rooted at V whose support vector to its parent, P, is strictly positive for all "relevant" values, can be pruned. Relevancy can be defined recursively as follows: the relevant values of V are those values which are consistent with some relevant value of its parent, and the relevant values of the root, X, are those which are not known to be excluded by any outside-world-change, independent of any change to the assumptions.

To illustrate the contradiction resolution process, consider the network given in Figure 6(a), which is an extension to the network in Figure 1 (the constraint are strict lexicographic order along the arrows.) Variables X_1, X_6 and X_7 are assumption variables, with the current assumptions indicated by the unary constraints associated with them. The support messages sent by each variable to each of its neighbors are explicitly indicated. (The overall support vectors are not given explicitly.) It can be easily shown that the value a for X_3 is entailed and that there are 4 extensions altogether. Suppose now that a new variable X_8 and its constraint with X_3 is added (this is again a lexicographic constraint.) The value a of X_8 is consistent only with value

b of X_3 (see Figure 6(b)). Since the support for a of X_3 associated with this new link is zero, the new support vector for X_3 is zero and it detects a contradiction. Variable X_3 will now activate a subtree for contradiction resolution, considering only its value b as "relevant", (since, value a is associated with a "0" support coming from X_8 which has no underlying assumptions). In the activation process, X_4 and X_5 will be pruned since their support messages to X_3 are strictly positive. X_1 will also be pruned since it has only one relevant value c and the support associated with this value is positive. The resulting activated tree is marked by heavy lines in Figure 6(b). Contradiction resolution of this subtree will be initiated by both assumption variables X_6 and X_7, and it will determine that the two assumptions $X_6 = c$ and $X_7 = c$ need to be replaced with assuming d for both variables (the process itself is not demonstrated).

Once contradiction resolution had been terminated, all assumptions can be changed accordingly, and the system can get into a new stable state by handling those changes using support propagation. If this last propagation is not synchronized, the amount of message passing on the network may be proportional to the number of assumptions changed. If, however, these message updating is synchronized, the network can reach a stable state with at most two message passing on each arc. Figure 6(c) gives the new updated messages after the system had been stabilized.

6. Support propagation in acyclic networks

Acyclic constraint networks extend the notion of a tree-structured networks to those having constraints of higher arity. Extending the notion of constraint graphs to non-binary networks we define the **primal constraint graph** which consists of a node for each variable and an arc for each two variables related directly by at least one constraint. Alternatively, a general network may be represented by a **dual constraint graph,** (or a hypergraph) consisting of a node for each **constraint** and an arc for any two constraints that share at least one variable. The dual constraint graph give rise to an equivalent binary constraint network, where variables are the con-

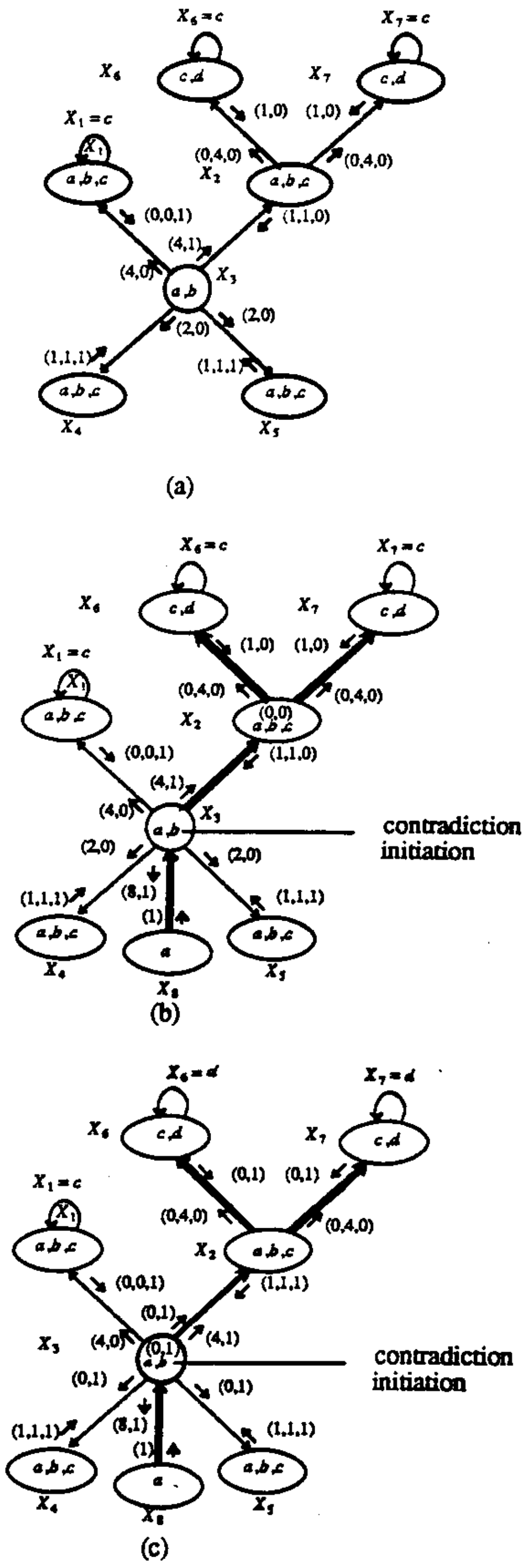


Figure 6. Illustration of the contradiction resolution process.

straints of the original network (called a c-variable), their tuples are their legal values and the constraints call for equality of the values assigned to the variables shared by any two c-variables.

For example, Figures 7(a) and 7(b) depict, respectively, the primal and the dual constraint-graphs of a network consisting of the variables A,B,C,D,E,F, with constraints on the subsets (ABC),(AEF), (CDE), and (ACE) (the constraints themselves are not specified).

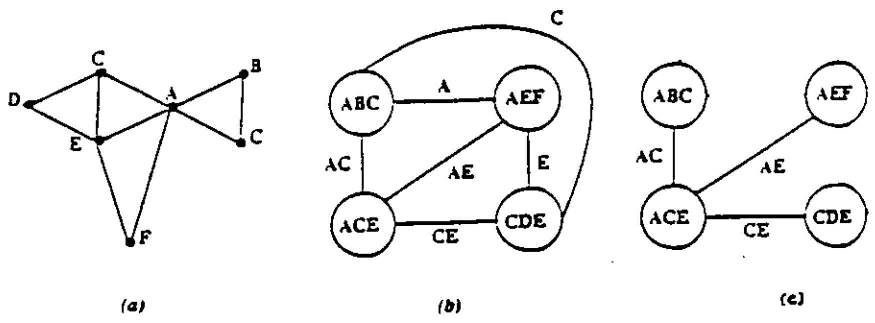


Figure 7: A primal and dual constraint graphs of a CSP

Since all the constraints in the dual representation are equalities, any cycle for which all the arcs share a common variable contains redundancy, and thus any arc such that each of the variables in its label is a common variable in some cycle may be removed from the network. The graph remaining after all such arcs have been removed is called a **join-graph**, and its corresponding network is equivalent to the original network.

For example, in Figure 7(b), the arc between (AEF) and (ABC) can be eliminated because the variable A is common along the cycle (AFE)—A—(ABC)—AC—(ACE)—AE—(AFE), so the consistency of the A variables is maintained by the remaining arcs. Similar arguments can be used to show that the arcs labeled C and E may be removed as well, thus transforming the dual graph into a **join-tree** (see Figure 7(c)). A Constraint network whose dual constraint graph can be reduced to a join-tree is said to be **acyclic**. Acyclic constraint networks are an instance of **acyclic data bases** discussed at length in [1].

The support propagation algorithm for tree-structured binary networks can be adapted for use in acyclic networks using one of their join-trees. We outline the algorithm next.

Consider the fragment of a join-tree, whose nodes represent the constraints C, U_1, U_2, U_3, U_4, given in Figure 8.

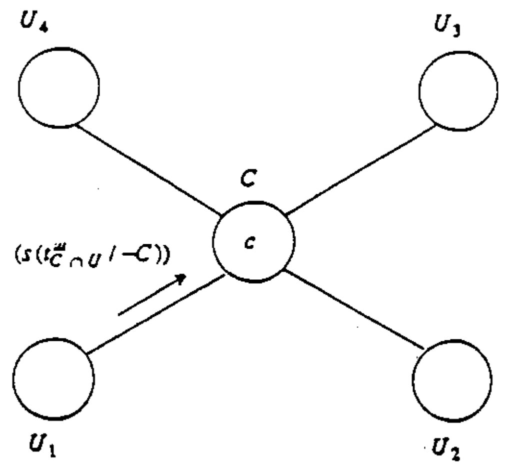


Figure 8: A fragment of a join-tree

We denote by t^c an arbitrary tuple of C. With each tuple, t^c, we associate a support number $s(t^c)$, which equals the number of solutions in which all values of t^c participate. Let $s(t^c \mid U)$ denote the support of t^c coming from subtree $T_{CU}(U)$, and let $s(t^c \mid -U)$ denote the support for t^c restricted to subtree $T_{CU}(C)$ (we use the same notational conventions as in the binary case). The support for t^c is given by:

$$s(t^c) = \prod_{U \in C\text{'s neighbors}} s(t^c \mid U) . \tag{7}$$

The support U contributes to t^c can be derived from the support it contributes to the projection of t^c on $C \cap U$, denoted by $t^u{}_{C \cap U}$, and this, in turn, can be computed by summing all the supports of tuples in U restricted to subtree $T_{CU}(U)$ that have the same assignments as t^c for variables in $C \cap U$. Namely:

$$s(t^c \mid U) = s(t^c{}_{C\cap U} \mid U) = \sum_{t^u{}_{C\cap U} = t^c{}_{C\cap U}} s(t^u \mid -C) \,. \tag{8}$$

Equations (7) and (8) yield

$$s(t^c) = \prod_{U \in C\text{'s neighbors}} \; \sum_{t^u{}_{C\cap U} = t^c{}_{C\cap U}} s(t^u \mid -C) \,. \tag{9}$$

The propagation scheme emerging from (9) has the same pattern as the propagation for binary constraint. Each constraint calculates the support vector associated with each of its outgoing arcs using:

$$s(t^u{}_{C\cap U} \mid -C) = \sum_{t^u{}_{C\cap U} = t^{u'}{}_{C\cap U}} s(t^{u'} \mid -C) \,. \tag{10}$$

The message which U sends to C is the vector

$$(s(t^{u_i}{}_{C\cap U} \mid -C)) \,, \tag{11}$$

where i indexes the projection of constraint U on $C \cap U$. Using this message, C can calculate its own support-vector (using (9)) and will also generate updating messages to be sent to its neighbors (using (10)).

Having the supports associated with each tuple in a constraint, the supports of **individual values** can easily be derived by summing the corresponding supports of all tuples in a constraint having that value.

Contradiction resolution can also be modified for join-trees using the same methodology. This process will be illustrated in the next section where these algorithms are demonstrated on a circuit diagnosis example. Support propagation and contradiction resolution take, on join-trees, the same amount of message passing as their binary network counterparts. Thus, the algorithm is linear in the number of constraints and quadratic in the number of tuples in a constraint. However, due to the special nature of the "dual constraints", being all equalities, the dependency of the complexity on the number of tuples t can be reduced from t^2 to $t\log t$, using a simple indexing technique.

7. A Circuit Diagnosis Example

An electronic circuit can be modeled in terms of a constraint network by associating a variable with each input, output, intermediate value, and device. Devices are modeled as bi-valued **assumption variables**, having the default value "0" if functioning correctly and the value "1" otherwise. There is a constraint associated with each device, relating the device variable with its immediate inputs and outputs. Given input data, the possible values of any intermediate variable or output variable is its "expected value", namely, the value that would have resulted if all devices worked correctly, or some ''unexpected
value'' denoted by "e". A variable may have more then one expected value. For the purpose of this example we assume that the set of expected values for each variable were determined by some pre-processing and all the other values are marked by the symbol "e".

Consider the circuit of Figure 9 (also discussed in [7, 2, 13]), consisting of three multipliers, M_1,M_2,M_3, and two adders, A_1 and A_2. The values of the five input variables, A, B, C, D, and F, and of the two output variables, F and G, are given. The numbers in the brackets are the expected values of the three intermediate points X, Y, and Z, and of the outputs. The relation defining the constraint associated with the multiplier M_1 is given in Figure 10 as an example, as well as the initial diagnostic weights associated with the tuples of these leaf constraints. The weight of the first tuple is "0" since the assumption variable, M_1 is assigned the currently assumed value, "0", while in the second tuple the assumed value is changed to "1". Given the inputs and outputs of the circuit, the objective is to identify a minimal set of devices which, if presumed to be malfunctioning, could explain the observed behavior (i.e., $G = 12$ and $F = 10$).

The dual graph of the constraint network corresponding to this circuit is given in Figure 11. This network is acyclic, as is evident by the fact that a join-tree can be obtained by eliminating the redundant arc (marked by a dashed line) between constraint (M_2,B,D,Y) and

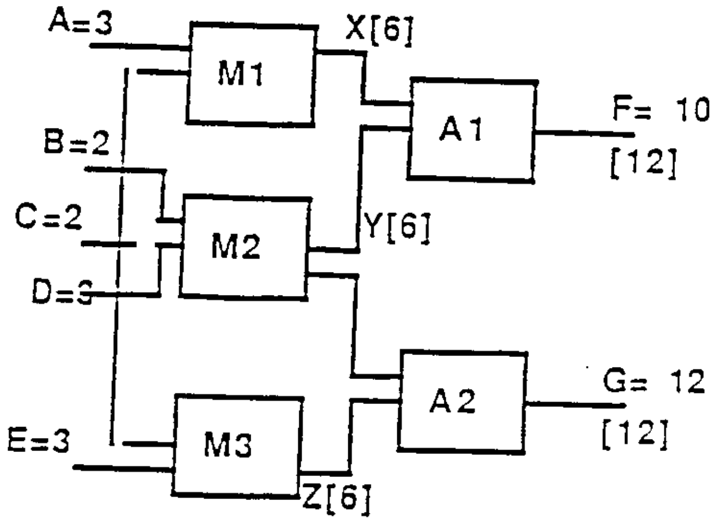


Figure 9: A circuit example

M_1	A	C	X	
0	2	3	6	$w=0$
1	2	3	ϵ	$w=1$

Figure 10: A multiplier constraint

(A_2,Z,Y,G).

Initially, when no observation of output data is available, the network propagates its support numbers assuming all device variables have their default value "0". In this case only one solution exists and therefore the supports for all consistent values are "1" (the support propagation algorithm is not illustrated). The diagnosis process is initiated when the value "10" is observed for variable F which is different from the expected value of 12. The value "10" is fixed as the only consistent value of F. At this point, the constraint (X,A_1,F,Y), which is the only one to contain F, induces direction on the join-tree, resulting in the directed tree (rooted at itself) of Figure 12, and contradiction resolution is initiated. Each tuple will be associated with the minimum number of assumption changes in the subtree underneath it, and the c-variable will

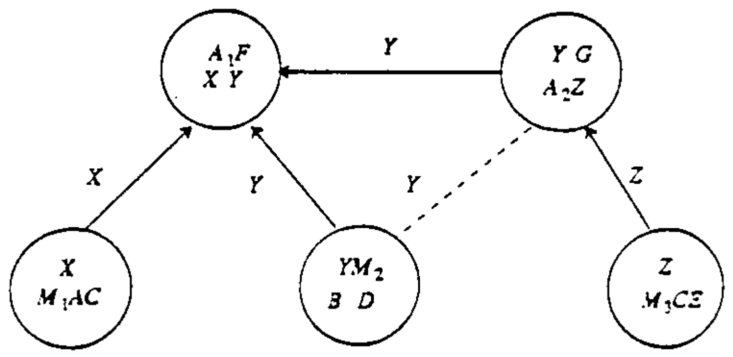


Figure 11: An acyclic constraint network of the circuit example

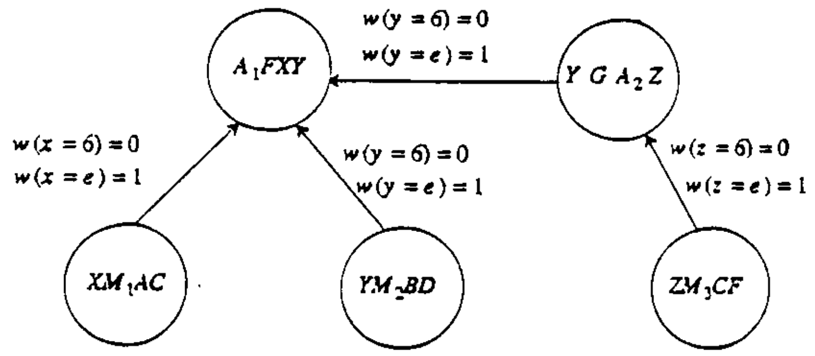


Figure 12: Weight calculation for the circuit example

project the corresponding weights on the variables which label its outgoing arc. In Figure 12 the weights associated with the arcs of the three leaf constraints (i.e., the multipliers constraints), are presented. They are derived from the weights associated with their incoming constraints (see the weights in Figure 10). For instance, the weights associated with X is $w(X=6)$=0 since "6" is the expected value of X when M_1 works correctly (which is the default assumption), and $w(X=e)=1$ since, any other value can be expected only if the multiplier is faulty. Next, the weights propagate to constraint (Y,G,A_2,Z). This constraint and its weights are given in Figure

13 (note, that G's observed value is 12).

A_2	Z	G	Y	Weights	Faulty Devices
0	6	12	6	$w = 0$	none
0	e	12	e	$w = 1$	M_3
1	6	12	e	$w = 1$	A_2
1	e	12	e	$w = 2$	M_3 & A_2

Figure 13: The weights of constraint (Y,G,A_2,Z)

The corresponding derived Y's weights are indicated on the outgoing arc of constraint (Y,G,A_2,Z) in Figure 12. Finally, the weights associated with the root constraint (A_1,X,Y,F) are computed by summing the minimum weights associated with each of its child node. The tuples associated with the root constraint and their weights are presented in Figure 14.

	A_1	F	X	Y	Weights	Faulty Devices
1.	0	10	6	e	2	$(M_3 \vee A_2)$ & M_2
2.	0	10	4	6	1	M_1
3.	0	10	e	e	3	M_1 & M_2 & $(M_3 \vee A_2)$
4.	1	10	6	6	1	A_1
5.	1	10	6	e	3	A_1 & M_2 & $(M_3 \vee A_2)$
6.	1	10	e	e	4	A_1 & M_2 & M_1 & $(M_3 \vee A_2)$

Figure 14: The weights of constraint (A_1,F,X,Y) (the root)

We see that the minimum weight is associated with tuples (2), indicating M_1 as faulty or (4), indicating A_1 as faulty. Therefore, either A_1 or M_1 are faulty. The weights can also be used as a guide for additional measurement taking in order to delineate between the different diagnoses.

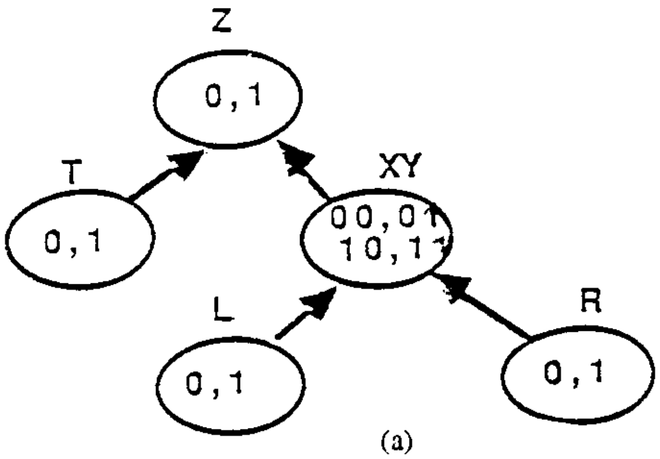
This example illustrates the efficiency of the contradiction resolution process when the special structure of the problem is exploited. See also [12]. By contrast, handling this problem using ATMS [7] exhibits exponential behavior. In a similar manner a propagation scheme can be devised to extract all minimal diagnoses (not necessarily the minimum-cardinality ones [5]) for further processing by some diagnostic package.

8. Finding all minimal supports in trees

Thus far we showed that the JTMS tasks are linear for singly connected networks (i.e. binary trees and their extensions to acyclic networks). In this and in the following sections we focus on the ATMS task as defined earlier for constraint networks. Namely, given a constraint network and a value x of X, the task is to find all minimal instantiations of assumption variables that supports $X=x$. Notice the distinction between **support sets** which are of concern for the ATMS task vs. the **support numbers** which were computed for the JTMS task. We will henceforth present an algorithm for computing the minimal support sets and we assume w.l.o.g. that all variables are assumption variables.

The basic idea of the algorithm is demonstrated through the example of figure 1. Suppose that all minimal supports for $Z=1$ are to be found. The algorithm generates a directed tree rooted at Z (figure 15a). Then, for each value, it computes all its minimal support sets **restricted to its child nodes.** For instance, the set of minimal support sets computed for $XY=11$ is $\{(L=1,R=1)\}$, while for $XY=01$ the set is empty since no instantiation of the child variables R and L entails $XY=01$. The subsets of minimal supports, restricted to children, are called **minimal support labels** . The minimal support labels of a value v of V, w.r.t. some directed tree T, is denoted by $msl_V(v)$ or $msl(v)$ when the identity of V is clear (see figure 15b).

Once all minimal support labels are computed, new supports can be generated by replacing a value in a label by one of its own minimal supports. For instance, since $(XY=11)$ is a minimal support label for $Z=1$ and since $XY=11$ is minimally supported by $(L=1,R=1)$, this past set is a new support for $Z=1$. This property seems to suggest that the set of supports can be generated by going from leaves to root recursively generating the support sets for each value (restricted to the subtree rooted at it).



(a)

For $\alpha \in \{0,1\}$

$$msl_L(\alpha)=\{\}$$
$$msl_R(\alpha)=\{\}$$
$$msl_{XY}(01)=\{\}$$
$$msl_{XY}(00)=\{\}$$
$$msl_{XY}(10)=\{\}$$
$$msl_{XY}(11)=\{(L=1,R=1)\}$$
$$msl_T(\alpha)=\{\}$$
$$msl_Z(1)=\{(T=1)(XY=01)(XY=10)(XY=11)\}$$

(b)

Figure 15

There are two problems with the above procedure. First, the minimality property **is not** maintained by this process; the set $(L=1,R=1)$, although generated by the substitution process, is not a minimal support for $Z=1$, since either $L=1$ or $R=1$ independently support $Z=1$. And an even more severe problem is that **not all support sets** are generated. Consider, for instance, the network in figure 15a restricted to variables (Z,XY,L). Each of the values $(XY=10)$ and $(XY=11)$ is a minimal support to $Z=1$. However, since each one of these values is not individually supported by $L=1$ ($L=1$ is consistent with both of them), $L=1$ cannot substitute either $XY=01$ nor $XY=11$, and will not be generated by the substitution process. Nevertheless, $L=1$ is a minimal support to $Z=1$.

To alleviate these two problems the algorithm that follows needs to make sure that all the support sets will be generated in the recursive substitution process. As was seen in the example, several values of a variable can play identical role in a support label (e.g., $XY=10$ and $XY=11$

both supporting $Z=1$) and therefore if a disjunction of such values has a minimal support, it can exchange any of the disjunct values in that label. For instance since $L=1$ supports the disjunction $XY=10$ *or* $XY=11$ it can exchange either one of them in a support. In the following paragraphs we formalize these notions.

The following notation sand definitions are needed. Let V and C be two variables having domains D_V and D_C, respectively, and let CV be the direct constraint between them. CV stands for the set of all legal pairs allowed by it. Let $M_{CV}(c)$ denote the set of values in D_V which are compatible with a value c of C (Figure 16a). Namely,

$$M_{CV}(c) = \{v \in D_V \mid (c,v) \in CV\}. \tag{12}$$

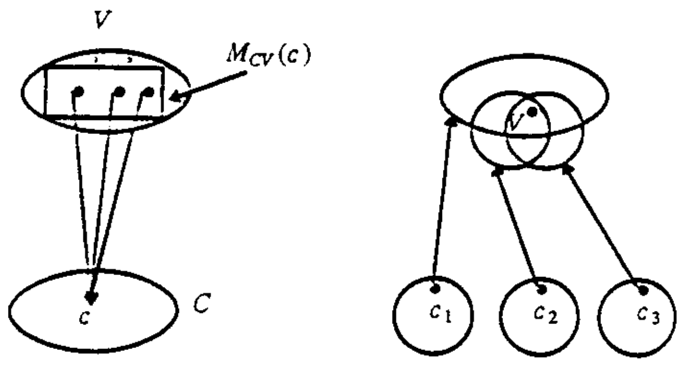
We first define the notion of support label, but instead of defining it for a singleton we define it for a subset of values in a variable's domain.

Given a variable V with its children $C_1, \ldots, C_l$ (Figure 16b) and a subset of consistent values A_V of V, a **support label** for A_V is an instantiation of a subset of the child variables, $(C_1 = c_1, \ldots, C_t = c_t)$, such that from all (the consistent) values of V A_V "matches" each of the child's values. Formally, the support labels satisfies

$$\bigcap_{c_j \in C_j} M_{C_jV}(c_j) \subseteq A_V. \tag{13}$$

A support-label is minimal if no subset of it satisfies condition (13). As we had seen, in the example of figure 15a, $\{Z = 1\}$ has four minimal support labels given by: $msl_Z(1) = \{(T = 1),(XY = 01),(XY = 10)(XY = 11)\}$. Another example given in figure 15a, is $msl_{XY}(\{01,10,11\}) = \{(L = 1),(R = 1)\}$ and it does not contain the support $(L = 1, R = 1)$ (which is not minimal).

Given a label, $l = (C_1 = c_1, \ldots, C_r = c_r, \ldots, C_t = c_t)$ that minimally supports a subset A_V of V we denote by $l \mid (C_r = c_r')$ the label resulting from exchanging c_r by c_r' in l. If $l \mid (C_r = c_r')$ is also a minimal support for A_V we say that c_r and c_r' play identical role w.r.t. l and



(a) (b)

Figure 16:

A_V and that they are exchangeable.

Given a subset A_V of V, a label $l \in msl(A_V)$, and a variable C_r in l we define the set of **label-dependent** values of C_r, denoted by $F_{A_V,l}(C_r)$, which are exchangeable in label l:

$$F_{A_V,l}(C_r) = \{c_r \mid l \backslash (C_r = c_r) \text{ is a minimal support to } A_V\} \quad (14)$$

A given minimal support label for A_V is also a minimal support set (remember all variables are considered assumption variables). Such a label can recursively generate additional minimal support sets. Let $l = (C_1 = c_1, \ldots, C_r = c_r, \ldots, C_t = c_t)$ and denote by $mss(A)$ all the minimal support sets for set A restricted to its rooted tree. The set of minimal supports generated via l in the subtree rooted at V, denoted by $mss_l(A_V)$, where l is a support for A_V, satisfies:

$$mss_l(A_V) = mss(F_{A_V,l}(C_1)) \times, \ldots, \times mss(F_{A_V,l}(C_r)) \times, \ldots, \times mss(F_{A_V,l}(C_t)) \quad (15)$$

and

$$mss(A_V) = \bigcup_{l \in msl(A_V)} mss_l(A_V) \quad (16)$$

Combining (15) and (16) yield a recursive equation for calculating the minimal support-sets of a given subset A_V of V which are restricted to its subtree:

$$mss(A_V) = \bigcup_{l \in msl(A_V)} \times_{\{C_i, C_i \in l\}} mss(F_{A_V,l}(C_i)) \tag{17}$$

The algorithm is described for a tree whose root is the queried variable V. The first phase is a top down process generating all the support labels of all the relevant label-dependent subsets. In this process, the root computes the support labels of the queried value, and then all the label-dependent subsets of each of its child nodes. Next, all minimal labels of each such subset are computed using EQ. (13) and they induce new label-dependent subsets on their own children. The process continues top down where the label-dependent subsets of a variable are computed only after their parents had already computed **all** their relevant support labels. The second phase involves a recursive generation of all the **support sets** via a bottom up substitution process that starts at the leaves and continues level by level until it reaches the queried variable. Each variable, in its turn, computes, for each of its label-dependent subsets, all its minimal support sets restricted to the subtree rooted at itself. This is accomplished by substituting a value c of C in a label l that supports A_V by one of the (already computed) minimal support sets of the subset $F_{A_V,l}(C)$.

A schematic description of the algorithm's steps is henceforth described. It assumes that the support numbers are explicitly maintained and thus the consistency of each value is known. Therefore, no inconsistent values participate in labels nor in label-dependent subsets.

Minimal-support-generation (V,{v})

```
1. begin
2.   generate-label-dependent subsets and their supports-labels.
3.   generate-supports-from-labels.
4. end
```

The following algorithm for generating the label-dependent subsets and their support labels is described for a parent variable V and its child nodes $C_1, C_2, \ldots, C_t$. Lets denote by $L(V)$ all the label-dependent subsets of V. It is assumes that V had already computed the minimal support labels for all his label-dependent subsets.

generate-label-dependent-subsets-and their-support-labels$(V, C_1, \ldots, C_t)$

```
1. begin
2.  for each $A_V \in L(V)$
3.   for each $l \in msl(A_V)$
4.    for each $C_i \in l$ do
5.      compute($F_{A_V,l}(C_i)$) and add $F_{A_V,l}(C_i)$ to $L(C_i)$
6.    end
7.   end
8.  end
9.  for each $C_i$ and for each $A \in L(C_i)$ do
10.   compute-minimal-labels($A$)
11. end.
12 end.
```

Performing the above procedure top-down, from root to leaves, will accomplish step 2 of the general algorithm. The computation of the label-dependent subsets (step 5) can be performed by scanning the labels of a given set A_V and determining, for each variable, a subset of its values that satisfies condition (14). The procedure **compute-minimal-labels**(A) (step 10) can be implemented by a standard search algorithm, that checks condition (13) for all instantiations of one child variable first, then goes to two variables, using values not selected previously, and continues to larger support labels. The support labels generated that way are minimal. The support labels of leaf values are the empty sets.

The algorithm for generating the minimal supports from labels is described for a subset A_V of a parent node V and all its child nodes $C_1, C_2, \ldots, C_t$.

Generate-support-from-labels($V, C_1, C_2, \cdots C_t$)

```
1. begin
2.  for each subset $A_V \in L(V)$ and for each label $l \in msl(A_V)$ do
3.    compute $mss_l(A_V) = \times_{C_i \in l} mss(F_{A_V,l}(C_i))$
4.  end
5.  for each $A_V \in L(V)$ compute
6.    $mss(A_V) = \bigcup_{l \in msl(A_V)} mss_l(A_V)$
7.  end
8. end
```

Theorem 1: Algorithm minimal-support-generation generates all and only minimal support sets.

Proof: It is clear that the algorithm generates only support sets. We will therefore focus on showing that any generated set is minimal and that any minimal support set is generated by the algorithm. The proof is by induction on the distance of the variables in the support set from the queried variable.

Let $Z=z$ be the queried variable and let $s=(X_1=x_1, \ldots, X_t=x_t)$ be a generated support set. We will show that s is a minimal support set. Let h be the the furthest distance from Z of variables in s. For $h=1$ it is known that only **minimal** support sets are generated, i.e., the support labels. Assuming that the generated supports having variables with distance $h-1$ or less are all minimal, we will show that s, having longest distance h, is also a minimal support set. Assume w.l.g. that $s^P=(X^P{}_{(1)}=x_{(1)}, \ldots, X^P{}_{(j)}=x_{(j)})$ is a partial sequence of the support variables in s having distance h and a common parent variable, P. Let $\bigcap_{x_{(i)} \in X^P{}_{(i)}} M_{X^P{}_{(i)}P}(x_{(i)})=A_P$.

Since s was generated by the algorithm, S^P must be a minimal support label of a label-dependent subset of P. Therefore A_P must be a subset of a label dependent subset of P and therefore each value of A_P can replace s^P in s, resulting in a support set with lower distance. By performing this "reverse substitution" to all the variables in s having distance h we get a support set whose utmost distance from Z is $h-1$ and which must have been generated by the algorithm. By the induction hypothesis this support is minimal and since each s_P is a **minimal** support label of the corresponding label-dependent subset, (otherwise it would not be applied) the substitution must result in a minimal support having distance h or less.

We will now show that if s is a minimal support set for $Z=z$ it must be generated by the algorithm. Let $\{s^{P_i}\}$ be all the subsets of s having distance h from Z indexed by their parents, P_i. Let the corresponding ranges that each set of children determines on their parent be defined by: $\bigcap_{x_{(j)} \in X^{P_i}{}_{(j)}} M_{X^{P_i}{}_{(j)}P_i}(x_{(j)})=A_{P_i}$. We claim that for every parent, P_i, each value in A_{P_i} can replace the subset s^{P_i} in s to yield a minimal support set of depth not greater then $h-1$, which we call s_{h-1}. Since, otherwise, if for some P_i and for some value in A_{P_i}, the resulting s_{h-1} is not a minimal support set for $Z=z$, it can easily be shown that from the definition of a support set, s could not be a minimal support set either, thus resulting in a contradiction. It follows that any possible s_{h-1}, generated by exchanging a value from A_{P_i} with S^{P_i} in s, is a minimal support set. Since s_{h-1} has distance $h-1$ at the most, the induction hypothesis implies that it is generated by the algorithm. Also, since by definition A_{P_i} is a label dependent subset of P_i, and since it is supported minimally by the label s^{P_i}, the algorithm will produce s in his substitution process.

□

9. Complexity analysis of the ATMS task

Following we discuss the time and space complexity of the ATMS algorithm.

9.1 Time complexity

The time complexity of the algorithm can be computed along its various steps. The computation of the minimal support labels and the label-dependent subsets is performed locally, between every node and its children. Given a parent node having d child variables and t label-dependent-subsets (already computed w.r.t. its parent), where $k \le t \le 2^k$, we can test condition (13) on subsets of child variables, in increasing order of their size. Since, in the worst-case, all subsets of $\frac{d}{2}$ variables may need to be tested, and since for every such set all combinations of values may be involved, we get:

$$T(label-generation) = \Theta\begin{pmatrix} d \\ \frac{d}{2} \end{pmatrix} \cdot k^{\frac{d}{2}} = \Theta(2\sqrt{k})^d. \tag{18}$$

This is the worst time performance that can be attained even when the number of support labels actually passing the test is very small and is, therefore, independent of the size of the result. Notice that in this case the tree structure **does not prevent exponential computation** of labels.

Once all labels are generated all label-dependent-subsets of each child variable will be generated and this may be of $O(t^2)$ (this step may require scanning the label set for each label).

Once the minimal support labels are available the time required for generating all minimal support sets is linear in the number of minimal support sets that will be generated since each substitution results in a minimal support set (see procedure generate-support-from-labels). Formally, if the number of the minimal support sets is n_s,

$$T(\mathit{supports-from-labels-generation}\ ) = \Theta(n_s). \tag{19}$$

The total time complexity of the algorithm is dominated, therefore, by the calculation of the minimal support labels and we get that the worst-case time complexity of algorithm *minimal support generation* is $T = \Theta(\exp(d) + n_s)$. Note that label generation is a special case of support set generation for trees with depth one hence it provides a worse-case lower bound on the inherent exponential complexity of the ATMS on trees.

9.2 Space complexity

The space complexity of the algorithm is the space required to store the minimal support labels and the label-dependent-subsets. Assume w.l.o.g. that the tree is uniform with degree d, that each variable has k values and that all minimal labels are of size r. We denote by n_l, the worst-case number of minimal support labels for an arbitrary label-dependent subset. Since there are at most $\binom{d}{r}$ variable-subsets of size r, and since every combination of elements from these subsets may generate a unique minimal support, the number of support labels, for all label-dependent subsets of a variable is bounded by

$$O(\binom{d}{r}\cdot k^r)). \tag{20}$$

Once more we can use $r = \frac{d}{2}$ as our worst-case. However, since each variable has only k values, the size of a minimal support label (or set) cannot exceed $k-1$. The reason is that each element-instantiation participating in a minimal support should decrease the intersection set (see (14)) by at least one value. We can, therefore, assume a worst-case situation in which the number of support labels are of size $r = min(k-1, \frac{d}{2})$, Substituting $r = min(\frac{d}{2}, k-1)$ in (20) and performing some simplification yields that the number of labels, n_l, has a worst-case bound of:

$$n_l = \begin{cases} \Theta(\frac{d^d}{(d-k+1)^{d-k+1}k}) & \text{, if } k-1 < \frac{d}{2} \\ \Theta((2\sqrt{k})^d) & \text{, if } k-1 \geq \frac{d}{2} \end{cases} \tag{21}$$

By substituting $k = \frac{d}{2}$ in (21) we get that the worst-case number of labels, n_l, has the following lower bound:

$$n_l \geq (2d)^{\frac{d}{2}}. \tag{22}$$

As to the number of minimal-support sets n_s, this number is not reduced by having a tree structure. Given n_l, the number of minimal support labels for each value, and assuming that all values have the same number of minimal support labels, we can compute the number of minimal support-sets for v in the subtree rooted at V. Let $n_s(i)$ be the number of support-sets of a value whose rooted tree has a depth i. Then $n_s(i)$ satisfies the following recursion:

$$n_s(i) = 1 + n_l(n_s(i-1))^r, \tag{23}$$

with

$$n_s(0) = 1.$$

From this recursion we get that:

$$n_s(i) = \begin{cases} O(n_l^{\frac{r^i-1}{r-1}}), & r>1 \\ \sum_{j=0}^{i} n_l^j = n_l^{i+1}, & r=1 \end{cases}. \tag{24}$$

Since the size of a label is bounded by $\min(k-1, d)$, and denoting the depth of the tree by h, we can conclude that the number of support-sets, n_s, of an arbitrary value is:

$$n_s = \begin{cases} O(n_l^{(k-1)^h}) = O(n_l^n) & r>1, k-1<d \\ O(n_l^{d^h}) = O(n_l^n) & r>1, k-1 \geq d. \\ O(n_l^h) & r=1 \end{cases} \tag{25}$$

For bi-valued variables the number of labels per variable and the time for their generation is reduced to d, and $n_s = O(d^h) = O(n)$.

10. The compiled approach

The minimal support sets can be computed in query time or can be pre-compiled. In view of the previous complexity analysis we propose that only the minimal support-labels and the label-dependent subsets will be compiled and maintained. The minimal support labels are space exponential in a much smaller exponent then the minimal support sets (see (21) and (25)) and they enable a linear time computation of all minimal support sets.

Since a query may involve any variable, any directed tree may be invoked and thus any partition of the neighbors of a variable into a parent node and child nodes is possible. We suggest that the tree should be maintained with some default directionality and that all computations be made w.r.t. it. When a query involving the root of the tree arrives, the minimal support-sets can be generated based on the stored minimal support labels. When a query concerning an arbitrary variable arrives, its directed tree will be created by changing the direction only on the path from it to the root, and as a consequence, only variables on this path will have to recompute their label-dependent subsets and their labels

When a new constraint arrives it may invalidate certain values or may present new supports, thus, resulting in two maintenance tasks: computing some new labels and maintaining the consistency of old labels. The complexity of computing only the **additional** minimal support labels of a value due to a new constraint between a new and an old variable, is in the worst case, as complex as recomputing all its minimal labels.

Since each change to the network invokes the propagation algorithm which updates the **support numbers**, and marks some values as inconsistent, the second task involves discarding support-labels containing elements whose values are inconsistent. This operation can be performed by each variable independently, and is linear in the size of the minimal support labels.

11. The case of Horn clauses

Restricting the knowledge to Horn clauses, as in ATMS, and the queries to single variable or a conjunction of variables but no disjunctions, corresponds to constraint-networks with bi-valued variable. (The concept of a three valued variable cannot be expressed by Horn clauses). In this case the minimal support labels, and the minimal supports have one value only, and thus, label computation becomes both time and space linear in the degree d of a variable and the number of support-sets is linear in n. This shows the tremendous computational savings presented by such a restriction.

When the Horn-clauses are not binary, but still maintain a singly-connected structure, they constitute an acyclic constraint network. The number of labels, in this case, is bounded by $\#c$, denoting the maximum number of constraints in which a variable participates. The number of minimal support sets, n_s, is $O(\#c^{(\#va-1)^h})$, when $\#va$, is the maximum number of variables in a constraint. In this case, therefore, the time and space complexity of the algorithm is linear while its output can still be exponential. As an example, consider the following set of Horn clauses:

$$(D \rightarrow C), (C, B \rightarrow A), (E, G, F \rightarrow C), (A, C \rightarrow E).$$

It can be represented by the dual constraint graph given in figure 17a having the "join-tree" in figure 17b.

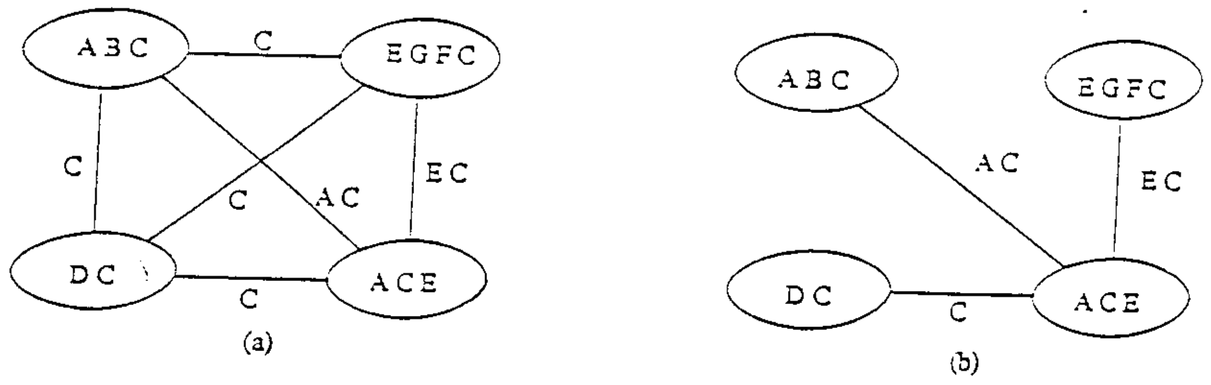


Figure 17

Consider the value $A=1$. A participates in two constraints (ABC,AEC) and only within (ABC), $A=1$ gets support. Its minimal support labels consist, therefore, of one set: ($C=1$,$B=1$). In order to compute all the minimal support sets for $A=1$, each value in a minimal label can be replaced by its own support coming from **outside the subtree** which contains A. The generation of the minimal support sets via the support labels is the same as in the binary case. The value $B = 1$ has no support except itself, while $C = 1$ is supported by its minimal labels $\{(D=1),(E=1,G=1,F=1)\}$ yielding four support sets to $A=1$ which are: $\{(A=1),(B=1,C=1),$ $(B=1,D=1),(B=1,E=1,G=1,F=1)\}$. Note that in this case there is no need to compute label-dependent subsets and it can be shown that for Horn clauses a simplified version of the algorithm suffices.

12. Summary and conclusions

In this paper we presented algorithms for truth-maintenance within the framework of constraint networks. The novel idea behind these algorithms is that they exploit the network's structure and results in a more efficient performance.

We presented two algorithms for the JTMS task, one for support propagation which determines the belief in each proposition and the other, contradiction-resolution, for consistency maintenance. Both algorithms are restricted to acyclic constraint networks for which they are time and space linear.

For the ATMS task the paper presents an algorithm for finding all minimal support sets for each proposition. The time and space complexity of this task is exponential, although the exponent is reduced from n -- the number of variables, to d -- the brunching degree of the graph. However, when restricted to Horn clause the ATMS task becomes linear thus justifying this common restriction in most ATMSs' implementations.

From these results it is apparent that the ATMS task is much more ambitious and much less tractable compared to the JTMS task even on singly-connected structures. Therefore the choice made by practitioners as to which approach to take must be carefully weighted against this tradeoff.

When the constraint network is not acyclic, the method of tree-clustering [6] can be used in a pre-processing mode. This method uses aggregation of constraints into equivalent constraints involving larger clusters of variables in such a way that the resulting network is acyclic. The complexity of the clustering scheme is exponential in the size of the largest cluster.

The applicability of the algorithms we presented is particularly useful for cases involving minor topological changes. This is the case when the knowledge-base models a physical or a biological system whose structure is fixed while changes occur via value manipulations only. In such cases the structure of the acyclic network, which may be compiled initially via tree-clustering, does not change.

References

[1] C. Beeri, R. Fagin, D. Maier, and M. Yannakakis, ''On the Desirability of Acyclic Database Schemes,'' *Journal of ACM,* Vol. 30, No. 3, 1983, pp. 479-513.

[2] R. Davis, ''Diagnostic Reasoning Based on Structure and Behavior,'' *Artificial Intelligence* , Vol. 24, 1984.

[3] R. Dechter and J. Pearl, ''Network-based Heuristics for Constraint-Catisfaction Problems,'' *Artificial Intelligence,* Vol. 34, No. 1, 1987, pp. 1-38.

[4] R. Dechter and J. Pearl, ''The cycle-cutset method for improving search performance in AI applications,'' in *Proceeding of the 3rd IEEE on AI Applications,* Orlando, Florida: 1987.

[5] R. Dechter, ''Constraint-Directed approach to diagnosis,'' UCLA, Cognitive systems Lab, Los Angeles, California, Tech. Rep. R-72-I, 1988.

[6] R. Dechter and J. Pearl, ''Tree Clustering for Constraint Networks,'' in *Artificial Intelligence,* 1989, pp. 353-366.

[7] J. de-Kleer and B. Williams , ''Reasoning about multiple-faults,'' in *Proceedings AAAI-86,* , Phila, PA.: 1986, pp. 132-139.

[8] J. de-Kleer, ''An Assumption-bBsed TMS,'' *Artificial Intelligence,* Vol. 28, No. 2, 1986.

[9] J. de-Kleer, ''A comparison of ATMS and CSP techniques,'' in *Proceedings of the 11th Intl. Conf. on AI (Ijcai-89,* Detroit: 1989, pp. 290-296.

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

[11] E.C. Freuder, ''A Sufficient Condition for Backtrack-Free Search.,'' *Journal of the ACM,* Vol. 29, No. 1, 1982, pp. 24-32.

[12] H. Geffner and J. Pearl, ''An improved constraint-propagation algorithm for diagnosis,'' in *Proceedings Ijcai,* Milano, Italy: 1987.

[13] M.R. Genesereth, ''The Use of Design Descriptions in Automated Diagnosis,'' *Artificial Intelligence,* Vol. 24, 1984, pp. 411-436.

[14] A. K. Mackworth, ''Consistency in Networks of Relations,'' *Artificial intelligence,* Vol. 8, No. 1, 1977, pp. 99-118.

[15] A.K. Mackworth and E.C. Freuder, ''The Complexity of Some Polynomial Network Consistency Algorithms for Constraint Satisfaction Problems,'' *Artificial Intelligence* , Vol. 25, No. 1, 1984.

[16] D. A. McAllester, ''An outlook on truth-maintenance,'' MIT, Boston, Massachusetts, Tech. Rep. AI Memo No. 551, 1980.

[17] J. Pearl, ''Fusion Propagation and Structuring in Belief Networks,'' *Artificial Intelligence* , Vol. 3, 1986, pp. 241-288.

[18] G. Provan, ''Complexity analysis of multiple-context TMSs in scene representation,'' in *Proceedings AAAI-87,* Seattle, Washington: 1987, pp. 173-177.

[19] R. Reiter and J. de-Kleer, ''Foundations of assumption-based truth maintenance systems: preliminary report.,'' in *Proceedings AAAI-87,* Seattle Washington: 1987.