Structure-Driven Algorithms for Truth Maintenance*

Rina Dechter
Information and Computer Science Department
University of California, Irvine , CA 92717
dechter@ics.uci.edu

Avi Dechter
Department of Managment Science
School of Business Administration and Economics
California State University, Northridge, CA 91330
avi@cs.ucla.edu

December 28, 1994

Abstract

This paper studies truth-maintenance and belief revision tasks on singly-connected structures for the purpose of understanding how structural features could be exploited in such tasks. We present distributed algorithms and show that, in the JTMS framework, both belief revision and consistency maintenance are linear in the size of the knowledge-base on singly connected structures. However, the ATMS task is exponential in the branching degree of the network. The singly-connected model, while restrictive, is useful for three reasons. First, efficient algorithms on singly-connected models can be utilized in more general structures by employing well-known clustering techniques. Second, these algorithms can serve as approximations or as heuristics in algorithms that perform truth-maintenance on general problems. Finally, the analysis provides insights for understanding the sources of the computational difficulties associated with JTMS and ATMS.

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 be consistent with each other and with the available information. When new facts become known, some assumptions must be changed so that the consistency of our view of the world is maintained at all times.

*This work was supported in part by grants from the Air Force Office of Scientific Research, AFOSR 900136, by NSF grant IRI-91573636, by grants from Toshiba of America and Xerox Palo Alto research center.

Truth-maintenance systems (TMSs) are computational schemes aimed at handling such situations. In its generic form, a TMS manipulates propositional formulas built from propositional symbols and standard Boolean connectives. A selected subset of the propositional symbols are called premises. That is, statements about the environment that do not require proof either because they are known to be true (observational facts) or because they may be assumed to be true as long as there is no evidence to the contrary (assumptions). In addition to premises, the system contains a set of boolean *constraints*, sentences which describe invariant properties of the environment.

The main functionality of a TMS is to determine whether the truth of a given proposition follows logically from a given set of premises and from the set of constraints, and to keep this information current.

Two primary approaches to TMS implementation have been proposed: the JTMS (Justification-based TMS) [13, 20, 22] and the ATMS (Assumption-based TMS) [5]. Each design implements the main functionality in a different way. A JTMS starts with the given premise set and attempts to identify all provable propositions, hoping that a proof will be derived for the target proposition. An ATMS, on the other hand, maintains for each proposition a collection of consistent premise subsets (called environments) any of which is sufficient for proving the proposition. To prove that the target proposition follows from the given premise set, all that is needed is to verify that this premise set contains at least one of the proposition's environments.

A JTMS must also be able to check whether its current premise set is consistent with the constraints, and, in case it is not, point to a part of the premise set which can be shown to be a source of the inconsistency. This functionality of JTMS is closely related to the framework of agent's belief revision. The basic idea in this more recent work is to enforce minimal change in an agent's belief necessary to account for a new contradiction in his knowledge [1]. Thus far, research in belief revision focused on the task of finding a minimal revision or on finding what holds under all minimal revisions. Here we focus on the identification of a *minimum number* of changes to represent minimal change.

When originally introduced, algorithms for Truth Maintenance were not accompanied by complexity analysis, or performance guarantees [13, 6, 20]. Nevertheless, experimental work with these tools, and more recent complexity analysis, have shown that both JTMS and ATMS functionalities are very inefficient, with ATMS exhibiting higher complexity than JTMS in both time and space.

A common strategy for reducing computational complexity has been to use efficient algorithms [21] which are complete only for restricted languages (e.g., unit resolution for Horn theories), but may be incomplete in general. This paper examine another type of restriction, one based on the *structure of the knowledge-base* as reflected in the graphical properties of the TMS constraint set. We will present algorithms that are tractable for tree-like knowledge-bases and whose complexity for general theories can be bounded as a function of the "distance" of the knowledge-base from a tree.

The algorithms are based on recent work in *Constraint Satisfaction Problems (CSPs)*, which resulted in many efficient algorithms and tractable cases tied specifically to the struc-

ture of the problem [11, 15, 19, 12]. A constraint network consists of a set of variables, each associated with a finite set of possible values, and a set of constraints, specifying joint assignments to the variables "allowed" by the constraints. A solution is an assignment of values to all the variables such that all of the constraints are satisfied.

For the purpose of using CSP techniques, we assume that the TMS knowledge-base is represented as a constraint network. In this network, each proposition (e.g., assumptions or facts) is represented by the assignment of a specific value to a particular variable. A proposition is entailed if it is the only consistent assignment of a value to that variable. To model change we introduce the notion of *assumption variables*. The TMS algorithms will be allowed to manipulate those assumptions in response to changes to the network (possibly imposed by observations from the outside world). Since the language of constraints has the same expressive power as propositional logic [20], all the algorithms presented here are applicable to propositional languages. The connection between truth maintenance systems and constraint satisfaction problems was already pointed out by several authors e.g., [6] and [23] [21]. The main thrust of these efforts has been to show that search reduction techniques developed in one area may be used to the benefit of the other. Our work here take this idea one step further.

We present the algorithms in a distributed fashion, that is, assigning a processor to each variable and letting the processors communicate with each other through a uniform protocol. We emphasize the distributed nature of our algorithms for two reasons. First, in order to allow their implementation on a real physical distributed network of processors. Second, a distributed algorithm, if *self stabilized*, is guaranteed to converge to a solution from any initial configuration (for precise definition see [25]). The algorithms we present are self-stabilized and, consequently, the updating process of such knowledge, in response to a local change, is already encoded in the protocol.

Following preliminaries in section 2 whereby the tasks of TMSs is defined within the constraint network model, we present (section 3) a distributed algorithm for computing and maintaining the *entailment* status of each proposition. In section 4 we present a distributed belief revision algorithm for restoring consistency to an inconsistent network in a way that minimize the *number of assumption changes*. We demonstrate the applicability of this algorithm for diagnosis (section 5). We show that, JTMS's tasks of entailment and belief revision are linear time on tree-like knowledge-bases. Finally, in section 6 we address the ATMS *labeling task* of compiling and maintaining all the labels. We show that, although in this case the algorithm is not tractable, not even for trees, its complexity is exponentially bounded by the branching degree of the tree. Therefore, chain-like networks can be processed relatively efficiently for this task as well.

2 Definitions and preliminaries

In this section we define the notions of *constraint networks* and *relations*, relate them to propositional theories, and define in their context the TMS tasks discussed in the introduction.

Definition 1 (relations, networks, schemes) *Given a set of variables* $X = \{X_1, ..., X_n\}$, *associated, respectively, with domains of discrete values* $D_1, ..., D_n$, *a* relation *(or, alternatively, a* constraint*)* $\rho = \rho(X_1, ..., X_n)$ *is any subset*

$$\rho \subseteq D_1 \times D_2 \times ... \times D_n.$$

The projection *of* ρ *onto a subset of variables* Q, *denoted* $\Pi_Q(\rho)$ *or* ρ_Q, *is the set of all tuples defined on the variables in* Q *that can be extended into tuples in* ρ. *A* constraint network R *over* X *is a set* $\rho_1, \ldots, \rho_t$ *of such relations. Each relation* ρ_i *is defined on a subset of variables* $S_i \subseteq X$. *The set of subsets* $S = \{S_1, .., S_t\}$ *is called the* scheme *of* R. *The network* R *represents a unique relation* $rel(R)$ *defined over* X, *which stands for all consistent assignments (or all solutions), namely,*

$$rel(R) = \{x = (x_1, ..., x_n) | \ \forall S_i \in S, \Pi_{S_i}(x) \in \rho_i\}.$$

A partial assignment $T = t$ *is a value assignment to a subset of variables* $T \subseteq X$. *A value* x *in the domain of* X *is said to be consistent if it is part of at least one solution of* R. *A tuple* t *is consistent if it participate in at least one solution.*

The scheme of a constraint network can be associated with a *constraint graph* where each subset in the scheme is represented by a node in the graph and two nodes are connected if the corresponding relations have at least one common variable. The arcs are labeled by the common variables. A networks whose constraint graph is a tree is said to be *acyclic* and its corresponding constraint graph is called a *join tree*. When all the constraints involve exactly two variables, the network is called *binary*. In such a case, another graphical representation of the network, called the primal constraint graph is useful. In this graph each variable is represented by a node and two nodes are connected by an arc if the variables they represent are joined by a constraint.

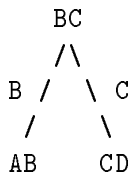
Any propositional theory can be viewed as a special kind of constraint network, where the domain of each variable is $\{0, 1\}$ (corresponding to $\{\mathbf{false}, \mathbf{true}\}$) and where each clause (a disjunction of propositional symbols or their negations) specifies a constraint (in other words, a relation) on its propositional symbols. The set of all models of the theory corresponds exactly to the set of all solutions of its corresponding constraint network.

A proposition $X = x$ is *entailed* by the network if x is the only consistent value of X, namely if it participates in all solutions. Similarly, a tuple t is entailed if it participate in all solutions. We distinguish a subset of variables $A \subseteq X$ called assumption variables.

Example 1 *Consider the theory* $\Phi = \{\neg A \vee \neg B, \neg B \vee \neg C, C \vee D\}$. *This theory can be viewed as a constraint network over the variables* $\{A, B, C, D\}$, *where the corresponding relations are the truth tables of each clause, that is,* $\rho(AB) = \{00, 01, 10\}$, $\rho(BC) = \{00, 01, 10\}$, *and* $\rho(CD) = \{01, 10, 11\}$. *The scheme of the theory* Φ *is* $\{AB, BC, CD\}$. *The set of all solutions to this network (and hence the set of models of* Φ*) is*

$$\rho(ABCD) = \{0001, 0010, 0011, 0101, 1001, 1010, 1011\}.$$

It is evident that none of the propositions or their negations is entailed. Theory Φ *is acyclic as demonstrated by its constraint graph:*



We next define formally the TMS tasks that we consider in this paper:

Definition 2 (TMS functionalities defined for constraint networks) *Given a network of constraints R, over a set of variables $X = X_1, ..., X_n$, with a subset A, $A \subseteq X$ of assumption variables, and a set of constraints $\rho_1, ..., \rho_m$ over X we define:*

- **JTMS main functionality:**, *Given an instantiation of a subset of the assumption variables, determine for each $X = x$ if it is entailed.*
- **JTMS belief revision:** *Given a set of assumptions for which the network is inconsistent, determine a minimal-size set of assumption change that restore consistency.*
- **ATMS main functionality***: For each value x of each variable X, determine the set of all consistent minimal (in the sense of set inclusion) instantiations of assumption variables that entail $X = x$.*

The algorithms we discuss in this paper assume, initially, that the theories are *acyclic*. These algorithms are extensible to arbitrary theories via a procedure known as *tree-clustering* [12], which compiles any theory into a tree of relations. Consequently, given a general theory, the algorithms presented in the sequel work in two steps: A join-tree is computed by tree-clustering, and then a specialized tree-algorithm for the particular TMS function is applied. The complexity of tree-clustering is bounded exponentially by the size of the maximal arity of the generated relations, and hence our algorithms are efficient for theories that can be compiled into tree networks of low-arity relations only. Examples of such theories are discussed in section 5. An alternative approach for extending the algorithm to cyclic theories is via a method known as *cycle-cutset* [9], which is exponential in the cycle-cutset size of the theory. The size of the minimal cycle-cutset is normally higher than the cluster's sizes of a tree-embedding of the same theory. The virtue of the cycle-cutset however is that, unlike tree-clustering, it does not require exponential space.

3 Identifying all entailed propositions

A proposition of the type $X = x$ is entailed in a network of constraint R if the network has at least one solution and if x is the only value of X in all solutions. The approach we propose for identifying all entailed propositions involves computing for each value in the domain of each variable the number of solutions it participates in. We will refer to this number as the *numerical-support* (*n-support* in short) of the value. Once this information is available for

every value of a variable, entailment is easy to determine locally: *a proposition $X = x$ is entailed if x has a positive n-support while all other values of X have zero n-supports.*

The notion of numerical support may be extended to encompass not just the instantiation of a single variable but that of a tuple of variables. This is captured in the following definition.

Definition 3 *Given a network of constraints over variables $X_1, ..., X_n$, with constraints $C_1, ...C_r$, the n-support of a value x of X, denoted $s_X(x)$, is the number of solutions in which X is assigned the value x. The n-support of a tuple $C_i = c_i$, denoted $s_{C_i}(c_i)$, is the number of solutions in which $C_i = c_i$. Let R be a join tree, and let (U,V) be an arc in the tree (see Figure 1). The subtree rooted at U which does not contain V, is denoted T_U^V, and $s_U^V(u)$ is the number of solutions of T_U^V which are consistent with $U = u$.*

In the following we present a *distributed* algorithm that compiles all n-supports for an acyclic network. The algorithm is a distributed adaptation of a known tree algorithm that computes the number of solutions [11]. Following [11], it is easy to show that:

Theorem 1 *Let T be a join-tree, and let U be one of its relations, the overall n-support for $U = u$ can be expressed as a function of the n-supports of its neighbors, namely:*

$$s_U(u) = \prod_{(Q,U)\in T} \sum_{q; q_{U\cap Q} = u_{U\cap Q}} s_Q^U(q) \tag{1}$$

□.

Equation (1) lends itself to a distributed propagation scheme. If constraint U gets from each neighboring node, Q, the n-support vector, s_Q^U, it can calculate its own support vector using equation (1), and, at the same time, it can generate an appropriate message for each of its own neighbors. The message which U sends to Q, s_U^Q, (i.e., the support vector reflecting the subtree T_U^Q,) can be computed by:

$$s_U^Q(u) \;=\; \prod_{(C,U)\in T, C\neq Q} \sum_{c; c_{C\cap U} = u_{C\cap U}} s_C^U(c) \tag{2}$$

The message generated by a leaf-constraint is a vector consisting of 1's representing the tuples allowed by that constraint. The computation consists of nodes sending to their neighbors the partial n-support vectors whenever they are readily computed. When all nodes have received all the n-supports, the overall n-supports for each tuple in their constraint can be computed using (1).

Having the n-supports for each *tuple* in each relation, the single-valued n-supports can be derived by picking a relation in the tree containing the variable in question, and then summing the corresponding n-supports of all tuples in the relation that has that value. The algorithm for node, U, is summarized in Figure 2.

If the algorithm is executed in a real distributed environment, its convergence is guaranteed after at most $n \cdot d$ messages when d is the the maximal distance between two leaf nodes. Under some synchronization the number of messages can be reduced to $2n$. It can, similarly be shown

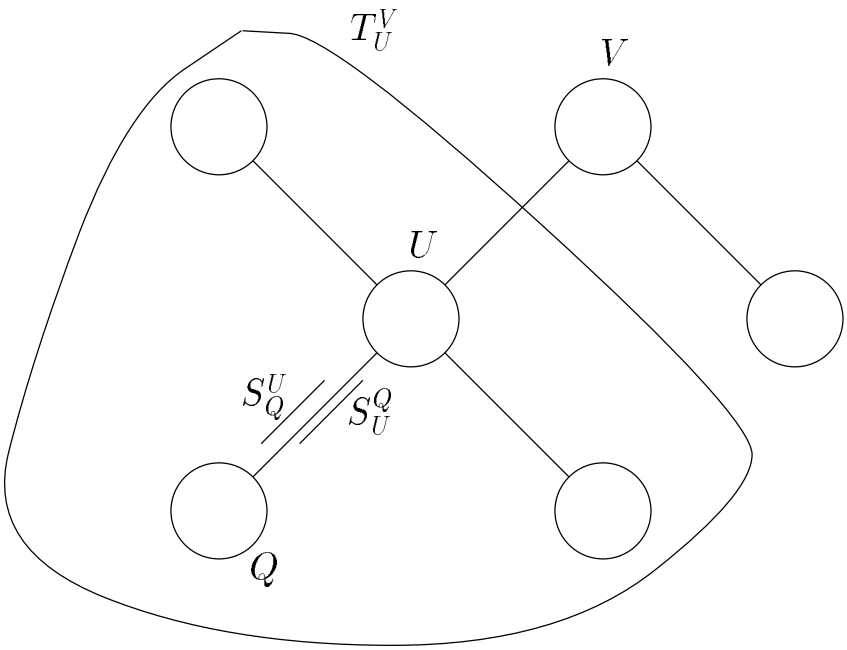


Figure 1: A fragment of a tree network

support propagation(U)
Input: A join-tree T, having variables $X = \{X_1, ..., X_n\}$. A relation $U \in T$, the n-support vector s_Q^U, for all neighbors, Q of U.
Output: The support vector $s_U(u)$, and for each neighbor Q, the vector $s_U^Q(u)$.

1. Compute n-supports for each tuple:

$$s_U(u) = \prod_{(U,Q)\in T} \sum_{q; q_{U\cap Q} = u_{U\cap Q}} s_Q^U(q)$$

2. Update single-value n-supports: For every X in the constraint U,

$$s_X(x) = \sum_{u\in U; u_X = x} S_U(u)$$

3. Compute n-supports messages for each neighbor Q:

$$s_U^Q(u) = \prod_{(C,U)\in T, C\neq Q} \sum_{c; c_{C\cap U} = u_{C\cap U}} s_C^U(c)$$

Figure 2: Algorithm support-propagation

that the cost of updating the n-supports following a single change in one relation will cause at most $2n$ messages until convergence. The following theorem focus only on the sequential complexity of the algorithm.

Theorem 2 *The sequential complexity of algorithm support-propagation is $O(n \cdot r \cdot logr)$, where r is the maximal number of tuples of any relation in the join-tree.*

Proof: The summation operation between any two relations U and Q can be accomplished in $O(r \cdot logr)$ steps as follows. Relation Q is projected on the variables in the intersection of Q and U. Each projected tuple is associates with a new n-support computed by summing the corresponding n-supports in the message that Q sends to U. This operation takes $O(r)$ steps. Then, the projected relation can be sorted in $O(r \cdot logr)$ steps and each tuple can be retrieved in $logr$ steps by relation U $\Box$.

In conclusion, once all n-supports are computed, each tuple of each relation "knows" if it is entailed, consistent, or inconsistent in constant time, while for each singleton this information can be obtained in $O(r)$ steps.

If one is interested in entailment only, *flat* support-vectors, consisting of zeros and ones, can be propagated in exactly the same manner, except that the summation operation in (1) should be replaced by the logic operator OR, and the multiplication can be replaced by AND. This results in a distributed arc-consistency algorithm that minimizes the number of message passing along the links.

Computing n-supports is not easier than determining entailment directly (namely, by calculating all the solutions of the network). In fact, it is generally very complex, as it is $\#P$-complete [24]. However, for acyclic networks the complexity of computing n-supports is the same as that of computing one solution. We choose to compute n-supports since this can be accomplished at no additional cost on trees, while it is more informative and might prove useful in applications.

4 Belief revision

When, as a result of a 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.

It is widely agreed that the subset of assumptions that are modified should be minimal, namely, it must not contain any proper subset of assumptions 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 find a minimum cardinality set of assumptions that need to change in order to restore consistency.

Assume that a constraint which detects an inconsistency (i.e., all its n-supports are zero) sends this information to the entire tree, creating in the process a directed tree rooted at itself. Given this rooted join-tree, belief revision proceeds as follows.

With each tuple v of each relation V in the join-tree T, we associate a weight $w(v)$, denoting the minimum number of assumption values that must be changed in the subtree

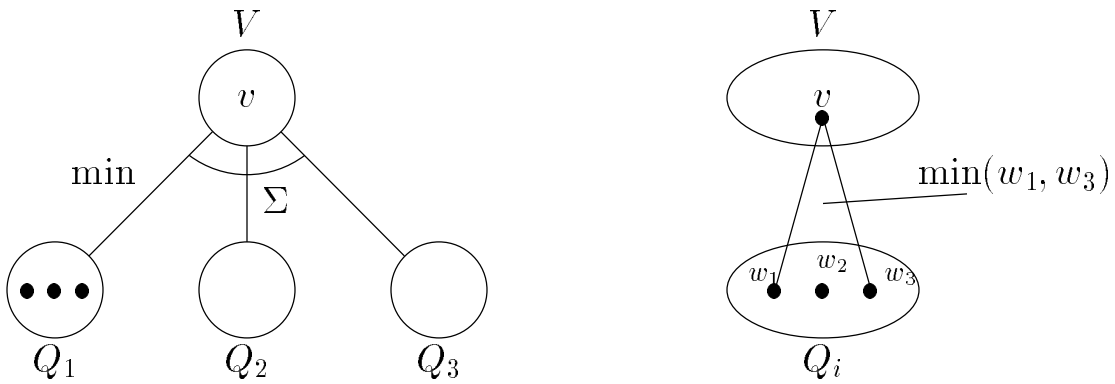


Figure 3: Weight calculation for relation V

.

rooted at V in order to make v consistent relative to this subtree. For each tuple q of a relation Q we denote by $\delta(q)$, the number of its assumptions that differ from the *current assumptions*. We denote by $child(V)$ the set of child relation of V in the join-tree.

The weights obey the following recursion(see Figure 3):

$$w(v) \;=\; \delta(v) + \sum_{Q;Q\in child(V)} \min_{q\in Q; q_{V\cap Q}=v_{V\cap Q}} w(q), \tag{3}$$

The computation of the weights is performed distributedly from the leaves of the directed tree to the root. A node waits to get the weights of all its child nodes, computes its own weights according to (3), and makes them available to its parent. During this *bottom-up-propagation* a pointer is kept from each tuple of V to the tuples in each of its child-nodes where a minimum is achieved. When the root receives all the weights, it computes its own weights and selects one of its minimizing tuples. It then initiates (with this tuple) a *top-down propagation* down the tree, following the pointers marked in the bottom-up-propagation. At termination this process marks the assumption variables that need to be changed and the appropriate changes required. The algorithms is summarized in Figure 4.

Theorem 3 *Given a join-tree T whose relation size is at most r, algorithm belief-revision can find one minimal revision in $O(n \cdot r \cdot logr)$. The set of all minimal cardinality revisions can be generated in $O(l + n \cdot r \cdot logr)$, where l is the size of the output.*

proof: The minimization operation between any two relations of size r can be accomplished in $O(r \cdot logr)$ steps as follows. A child node C, with its weighted relation, projects its relation on the intersection of its own variables and that of its parent relation P. The weight associated with each projected tuple is the minimum weight among the weights of all the corresponding tuples in C. This operation can be accomplished in linear time $O(r)$. Then

Belief-revision (V)
Input: A rooted join-tree T, having variables $X = \{X_1, ..., X_n\}$, and relations $V_1, , , V_m$. A set of assumption variables $A \subseteq X$ and its current assignment $A = a$.
Output: The weights of each tuple and an indication to changes in assumption assignment.

1. For each tuple $v_i \in V_i$ compute $\delta(v_i)$.
2. **(Bottom up)** compute weights of V (given the weights for it child nodes)

$$w(v) \;=\; \delta(v) + \sum_{Q;Q \;\in child(V)} \min_{q \in Q; q_{V \cap Q} = v_{V \cap Q}} w(q),$$

3. **(Top down)** Given a tuple t selected by parent, Change assumption value accordingly, select that tuple of each child relation pointed to by t.

Figure 4: Algorithm belief revision

the projected relation can be sorted in $O(r \cdot logr)$ steps and can be retrieved in $logr$ steps by the parent node P for the summation operation. If all the minimum cardinality revisions are needed they can all be retrieved in output linear time by following the pointers in the top-down step of the algorithm. □

Once belief revision had been terminated, all assumptions can be changed accordingly, and the system can get into a new stable state using support propagation. There is no need, however, to activate the whole network for belief revision, because the n-support information clearly points to those subtrees where no assumption change is necessary. We will illustrate the belief revision algorithms in the next section using a circuit diagnosis example.

5 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 value "0" if functioning correctly (default) 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 collectively by the symbol "e".

Consider the circuit of Figure 5 (also discussed in [7, 4, 17]), 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 E, 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.

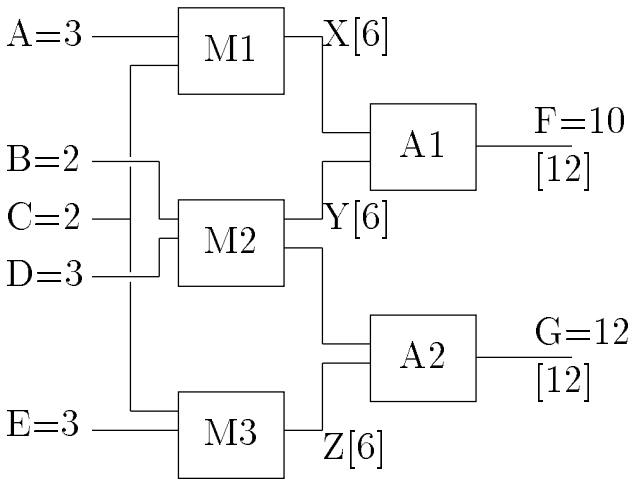


Figure 5: A circuit example

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

Figure 6: A multiplier constraint

The relation defining the constraint associated with the multiplier M_1 is given in Figure 6 as an example, as well as the initial weights associated with the tuples of these leaf constraints ($\delta = w$ for leaf nodes). 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 be consistent with the observed behavior (i.e., $G = 12$ and $F = 10$).

The join-tree of the constraint network modeling this circuit is given in Figure 7. 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 (A_2, Z, Y, G). For more details see [12].

Initially, when no observation of output data is available, the network propagates its n-supports assuming all device variables have their default assumption-value "0". In this case only one solution exists and therefore the supports for all consistent values are "1". 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)[1], which is the only one to contain F has all its

[1] For simplicity we will refere here to a constraint by the subset of variables on which it is defined

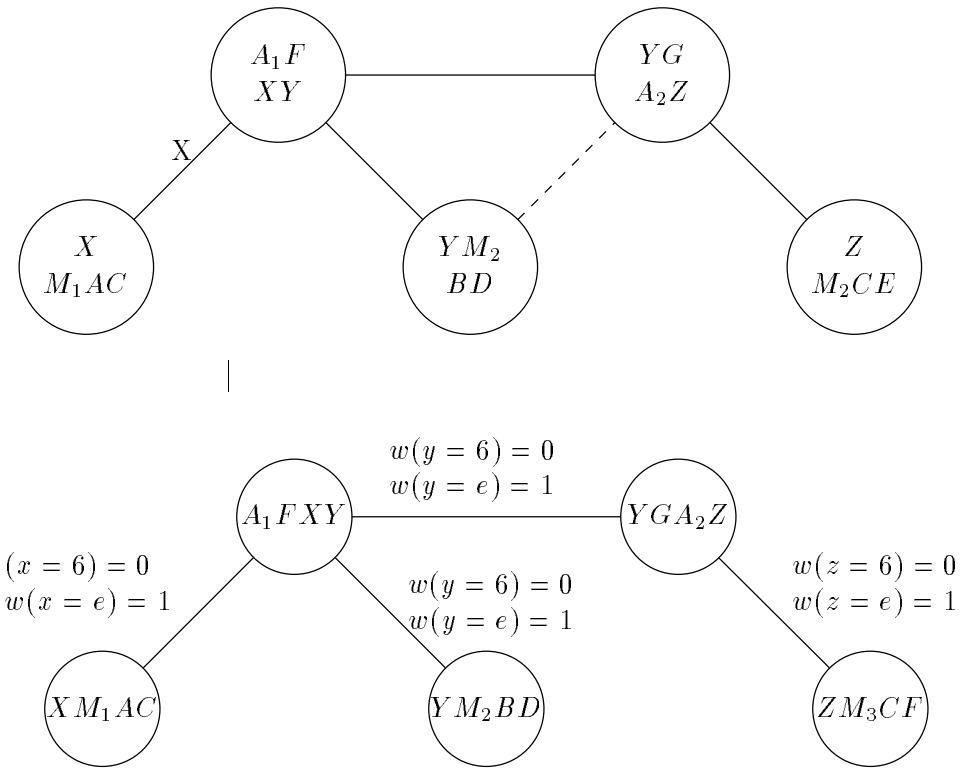


Figure 8: Weight calculation for the circuit example

n-supports equal "0" and it induces direction on the join-tree, resulting in the directed tree (rooted at itself) of Figure 7, and belief revision is initiated.

Each tuple will be associated with the minimum number of assumption changes in the subtree underneath it. Instead of indicating the weights associated with each tuple we indicate the weight projected on the variables on the arcs of the tree. In Figure 8 the weights associated with the arcs of the three leaf constraints (i.e., the multipliers constraints), projected on their outgoing arcs is illustrated. These are derived from the weights associated with their incoming constraints (see the weights in Figure 6). 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 is the only parent node of (Z, M_3, C, F) and its weights are given in Figure 9 (note, that G's observed value is 12).

The corresponding projected Y's weights are indicated on the outgoing arc of constraint (Y, G, A_2, Z) in Figure 8. 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 10.

We see that the minimum weight is associated with tuple (2), indicating M_1 as faulty

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

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

or tuple (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 that should delineate between the different diagnoses.)

This example illustrates the efficiency of the belief revision process when the special structure of the problem is exploited. By contrast, handling this problem using ATMS [5] may exhibit exponential behavior. A similar algorithm exploiting the framework of probabilistic networks is given [16].

6 ATMS Labeling

In this section we focus on the primary ATMS functionality, namely, finding one or all minimal instantiations of assumption variables in a given network of constraints that entail the proposition $X = x$. This task is often called *label determination* in the ATMS terminology. We call each tuple representing such an instantiation a *support tuple* or t-support. The main result of this section is a lower bound on the complexity of finding one t-support for $X = x$ which is exponential in the branching degree of the tree. We also introduce an algorithm for performing this task which attains this complexity, thereby proving that the bound is tight. When computing *all* t-supports, the complexity increases by a linear factor of the output. If only a subset of the variables are regarded as assumption variables, the complexity is still exponential in the degree of the tree unless the assumption variables are distributed in a way that reduces the *effective* degree of the tree. It should be noted that the ATMS task

is equivalent to what is often referred to as *abduction*, and, therefore, the results we present extend to the abduction task as well.

To simplify the exposition we will describe an algorithm for computing minimal t-supports for trees of *binary constraints*. In this case, the *primal constraint graph*, where nodes represent variables and arcs indicate the existence of constraints between pairs of variables, is more convenient. The extension of this algorithm to general join-trees is straightforward since a join-tree can be viewed as a regular binary tree where each relation is a compound variable and its tuples are its values. We assume, without loss of generality, that all the variables are assumption variables.

Definition 4 *Given a network of constraints R over a set of variables X, a partial instantiation $T = t$, $T \subseteq X$, is a support tuple (t-support) for $X_i = x_i$ iff for every solution s of R,*

$$s_T = t \ \rightarrow \ s_{X_i} = x_i. \tag{4}$$

$T = t$ is a minimal t-support of x_i is there is no subtuple of t satisfying (4). The set of all minimal t-supports for x_i relative to a network R is denoted $mst_R(x_i)$ or $mst(x_i)$ when the network's identity is clear.

Example 2 *Consider the following propositional theory*

$$\varphi = \{(T \rightarrow Z), (R \rightarrow Y), (L \rightarrow X)((X \vee Y) \rightarrow Z)\} \tag{5}$$

The theory can be modeled as binary tree network with four bi-valued variables T, Z, R, L and the compound variable XY having the domain $\{00, 01, 10, 11\}$ The explicit constraints are given by their truth tables. The constraint graph and the explicit constraints are depicted in Figure 11.

We will illustrate the main idea of the algorithm through an example. Suppose that we want to compute all minimal t-supports for $Z = 1$ in φ. The algorithm begins by generating a directed tree rooted at Z. Then, for each variable and each of its values, it computes all its minimal support tuples *restricted to the child nodes* of the relation, called *minimal child support*. For instance, the set of minimal child supports for $XY = 11$ is $(L = 1, R = 1)$, while for $XY = 01$ the set is empty since no tuple over the child variables R and L entails $XY = 01$. The set of all minimal child supports for $X = x$, in a given directed tree, is denoted by $mcs(x)$.

Given a minimal child support, l, new supports can be generated by replacing a value in l by one of its own minimal child supports. For instance, since $(XY = 11)$ is a minimal support tuple 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 support tuples can be generated recursively by a bottom up process from leaves to root.

There are, however, two problems. First, the minimality property *is not* maintained by this process. The set $(L = 1, R = 1)$, which has been generated by the substitution process, is not a minimal support for $Z = 1$ since either $L = 1$ or $R = 1$ independently support $Z = 1$.

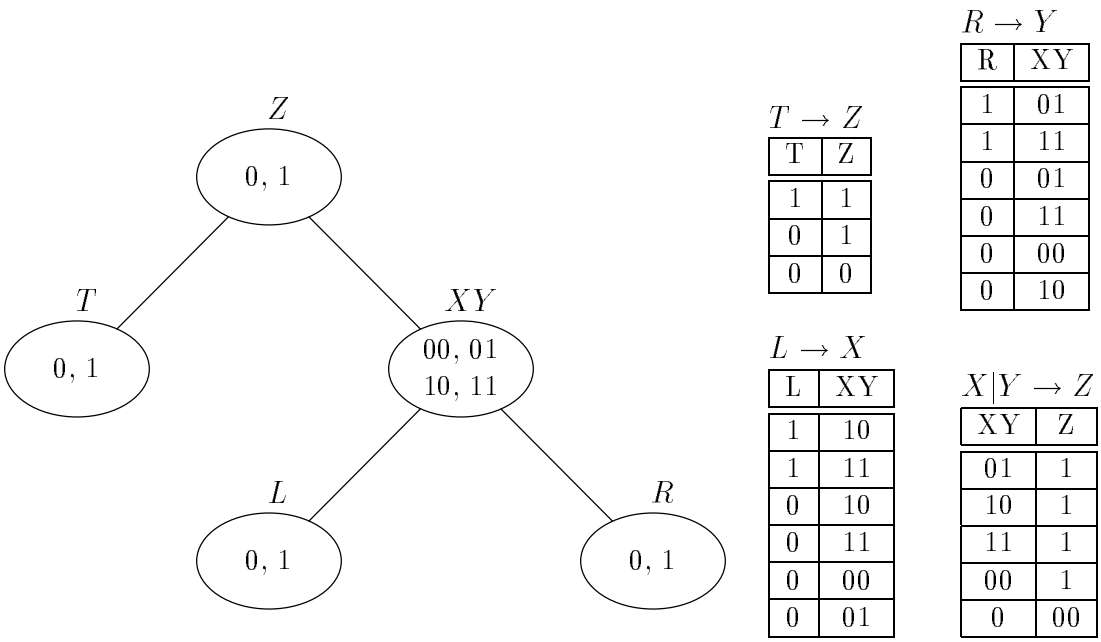


Figure 11: An example of a binary tree network

The second, and the more significant problem is that *not all support tuples* are generated. Consider again the tree in Figure 11a 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), $L = 1$ will not be generated by the substitution process (although it is a minimal support for $Z = 1$) because it can substitute neither $XY = 01$ nor $XY = 11$.

The algorithm we present next is base on the idea discussed above: compute the minimal t-supports locally and generate the rest by a recursive substitution process. To overcome the problems mentioned, the algorithm computes local t-supports to a *subsets of values* in a variable's domain, rather than to singletons. As was shown in the example, several values of a variable can play identical role in a support tuple (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 replace any of the elements of this disjunction in that label. For instance, since $L = 1$ supports the disjunction $XY = 10$ *or* $XY = 11$ it can replace either one of them is a t-support. Next, we formalize these notions.

Definition 5 *Let V and C be two variables in a network, and let R_{CV} denote the constraint of allowed pairs between them. $M_C^V(c)$ denotes the set of values in the domain of V that are consistent with $c \in C$ (Figure 12a). Namely,*

$$M_C^V(c) \;=\; \{v \in V |\ (c, v) \in R_{CV}\} \tag{6}$$

Definition 6 *Given a variable V with its child nodes $C_1, ..., C_l$ (Figure 12b) and a subset of*

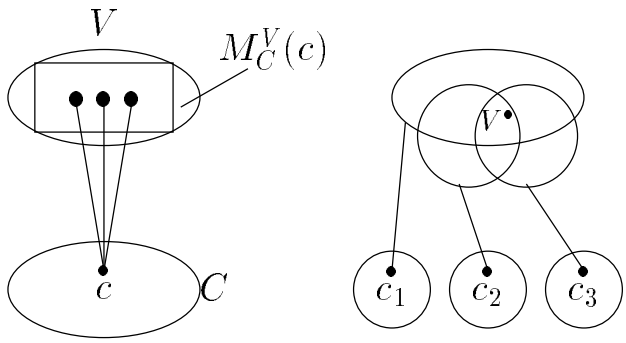


Figure 12: Computing the minimal child support

consistent values of V, denoted A_V, a child support for A_V is an instantiation tuple $(C_1 = c_1, ..., C_t = c_t)$ over a subset of its child variables, that entails A_V. Namely, a child support tuple, $(C_1 = c_1, ..., C_t = c_t)$, satisfies:

$$\bigcap_{c_j \in C_j} M_{C_j}^V(c_j) \subseteq A_V. \tag{7}$$

A child-support tuple is minimal if no subset of it satisfies condition (7).

Example 3 *It can be verified that (see 11), $Z = 1$ has four minimal child supports given by:*

$$mcs_Z(\{1\}) = \{(T = 1), (XY = 01), (XY = 10)(XY = 11)\},$$

while $mcs_{XY}(\{01, 10, 11\}) = \{(L = 1), (R = 1)\}$, and it does not contain the support $(L = 1, R = 1)$ since it is not minimal.

Given a minimal child support, $l = (C_1 = c_1, ..., C_t = c_t)$ of A_V, we denote by $l|(C_r = c_r\prime)$ the tuple resulting from exchanging c_r by $c_r\prime$ in l. If $l|(C_r = c_r\prime)$ is also a minimal t-support for A_V we say that c_r and $c_r\prime$ play identical role with respect to l and A_V and that they are exchangeable.

Definition 7 *Given a subset A_V of V, a tuple $l \in mcs(A_V)$, and a variable-value pair $C = c$ in l we define the set of* label-dependent *values of C, denoted by $F_{A_V,l}(C)$, (or $F_l(C)$ for short) which are exchangeable in label l:*

$$F_{A_V,l}(C) = \{c \in C |\ l|(C = c) \in mcs(A_V)\} \tag{8}$$

A minimal child support for A_V is also a minimal support since we assumed that all variables are assumption variables, and it can recursively generate additional minimal support tuples. Let $l \in mcs(A_V)$ such that

$$l = (C_1 = c_1, ..., C_r = c_r, ..., C_t = c_t)$$

label-generation (V)
Input: A tree T rooted at V. V has a set of child variables $C_1, ..., C_t$. The set of label dependent subsets of V, $L(V)$.
Output: The set of all minimal support sets to each $A \in L(V)$.

1. **top down:** for each $A_V \in L(V)$ compute the set of child support tuples $mcs(A_V)$ using equation (7).
2. for each $l \in mcs(A_V)$ and for each $c_i \in l$ do
3. compute $F_l(C_i)$ and add it to $L(C_i)$
4. **bottom up:**, (once $mst(F_l(C_i))$ are available) for each $A_V \in L(V)$ compute $mst(A_V) = \cup_{l \in mcs(A_V)} \times_{C_i \in l} mst(F_l(C_i))$

Figure 13: Algorithm minimal supports

and denote by $mst(A_V)$ all the minimal support tuples for A_V restricted to its rooted tree. The set of minimal supports generated via $l = (C_1, ..., C_t)$, in that subtree satisfies:

$$mst_l(A_V) = mst(F_l(C_1)) \times, ..., \times mst(F_l(C_r)) \times , ..., \times mst(F_l(C_t)) \tag{9}$$

and

$$mst(A_V) = \cup_{l \in mcs(A_V)} mst_l(A_V) \tag{10}$$

Combining (9) and (10) yields a recursive equation for calculating the minimal support tuples of a subset A_V restricted to the subtree rooted at V:

$$mst(A_V) = \cup_{l \in mcs(A_V)} \times_{\{C_i \in l\}} mst(F_l(C_i)) \tag{11}$$

The algorithm for generating all minimal support tuples is summarized in figure 13. The first phase is a top down process generating all the minimal *child-supports* of all the relevant *label-dependent* subsets. The process continues top down where the label-dependent subsets of a variable are computed only after their parents had already computed *all* their minimal child supports (or mcs's for short). The second phase is bottom up process that starts at the leaves and continues level by level until it reaches the queried relation. Each relation computes, in its turn, for each of its label-dependent subsets, all its minimal t-supports restricted to their tree. 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 $F_l(C)$.

The algorithm assumes that the n-supports are explicitly maintained and thus no inconsistent values participate in a support nor in a label-dependent subsets.

The computation of the label dependent subsets can be accomplished by scanning the set $mcs(A_V)$ of a given set A_V and determining, for each child node the subset of its values satisfies condition (8). Computing the minimal-child-supports for A_V can be implemented by a standard search algorithm that checks condition (7) for all instantiations of one child variable first, then goes to two variables, using values not selected previously, and continues to larger supports. The supports generated that way are minimal. The supports of leaf values are the empty tuples.

Theorem 4 *: Algorithm label-generation generates all and only minimal supports. (for proof see appendix).*

The time complexity of the algorithm can be computed along its various steps. The computation of the minimal-child-supports 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 with respect to its parent), where $k \leq t \leq 2^k$, we can test condition (7) 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:

$$T(label - generation) = \Omega \left(\begin{array}{c} d \\ \frac{d}{2} \end{array} \right) \cdot k^{\frac{d}{2}} = \Theta(2\sqrt{k})^d. \quad (12)$$

This performance can be attained in the worst-case even when the size of the resulting *mcs* set is very small. Notice that the tree structure *does not prevent exponential computation* for finding even one *mcs*.

Once all *mcs*'s for a variable are generated, all label-dependent-subsets of each child variable will be generated and this may take $O(t^2)$ steps.

Once the minimal child supports for all variables are available the time required for generating all minimal support tuples is linear in the output (this is the bottom up process). The total time complexity of the algorithm is dominated, therefore, by the calculation of the minimal child supports. Consequently,

Theorem 5 *The worst-case time complexity of algorithm has a lower bound* $\Omega(exp(d) + n_s)$*, and an upper bound* $O(exp(d) + n_s)$*, where* n_s *is the number of minimal support tuples in the output.* □

The results in this section are restricted for binary tree-networks. A simple way for extending them to an arbitrary join-tree is to view each relation in the tree as a meta variable when the constraints between relations say that two partial tuples are consistent if they coincide on shared variables. Once we compute for each tuple in the root relation all its minimal supports, the minimal supports of individual values can be generated by manipulating only the tuples of that relations which is at most exponential in the size of that relation. Consequently we can conclude that:

Corollary 6 *Finding all minimal support tuples of a partial tuple* t *appearing in a root relation of a join-tree can be accomplished in* $O(t^d + n_s)$ *when* t *bounds the number of tuples in each relation,* d *is the branching degree in the tree and* n_s *is the size of the output.* □

Corollary 7 *On chain-like join-trees one minimal* t*-support can be determined in* $O(exp(t))$ *when* t *is the maximum number of tuples in a relation.*

7 Conclusions and related work

This paper presents truth-maintenance algorithms within the framework of constraint networks. The common feature making these algorithms attractive is that their complexity is linked to the structure of the network, making their performance more predictable. Our results provide some theoretical explanations for the behavior of truth-maintenance algorithms reported in the literature.

The ideal structure for all three algorithms is an acyclic network. In that case, the two JTMS algorithms (entailment and belief revision) are time and space linear. On the other hand, the ATMS algorithm, which finds all minimal support sets for each proposition, is time and space exponential. The exponent is reduced though from n – the number of variables, to d – the branching degree of the constraint graph.

These results support the common observation that the ATMS task is more complex than the JTMS . It appears that, in practice, users should be advised to implement the JTMS strategy whenever possible and to use ATMS only when necessary.

When the constraint network is not acyclic, the method of tree-clustering [12] should be used as a pre-processing step. 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 clustering scheme is exponential in the *size of the largest cluster*, making the complexity of preprocessing also dependent on the structure of the network, with near-tree networks requiring less computation.

An alternative approach to dealing with cyclic theories is to use the *cycle-cutset* method [9], which is exponential in the cycle-cutset size of the theory. The size of the minimal cycle-cutset is larger than the cluster's sizes of a tree-embedding of the same theory, however, unlike tree-clustering, the cycle-cutset method does not require exponential space.

Because of the need for re-structuring the network, it is clear that the algorithms proposed in this paper are best-suited for situations where the structure of the knowledge-base is either fixed or involving only minor topological changes over time. Examples of such cases are physical or biological systems such as electronic circuits and model-based medical diagnosis systems.

Structure-exploiting algorithms for diagnosis that were inspired by algorithms on probabilistic networks appear in [16] and recently in [3, 18]. The merits of the belief revision algorithms appearing here were recently tested experimentally on various circuit diagnosis examples. The algorithm's performance was shown to be superior relative to Model-based Diagnosis (MBD) algorithms [14]. Another recent paper [2] discusses the task of finding all minimal sets of changes needed to restore consistency (as opposed to the task of all sets of minimum cardinality, discussed in the current paper). It is shown that while a propagation scheme is available in this case as well, it has greater complexity.

References

[1] C.R. Alchourron, P. Gardenfors, and D. Makinson. On the logic of theory change: partial meet contraction and revision functions. *Journal of Symbolic logic*, 50:510–530, 1985.

[2] R. Ben-Eliyahu and R. Dechter. On computing minimal models. In *Proceedings of AAAI-93*, pages 2–8, Washington, D.C., 1993.

[3] A Darwiche. Conditional independence in atmss: Independence-based algorithms for computing labels and diagnosis. In *Rockwell, Technical Report*, 1994.

[4] R. Davis. Diagnostic reasoning based on structure and behavior. *Artificial Intelligence*, 24, 1984.

[5] J. de Kleer. An assumption-based TMS. *Artificial Intelligence*, 28(2):127–162, 1986.

[6] J. de Kleer. A comparison of ATMS and CSP techniques. In *Proceedings of IJCAI-89*, pages 290–296, Detroit, Michigan, 1989.

[7] J. de Kleer and B. Williams. Reasoning about multiple faults. In *Proceedings of AAAI-86*, pages 132–139, Philadelphia, Pensilvania, 1986.

[8] R Dechter. A distributed algorithm for atms. In *Bar-Ilan Symposium on Foundation of Artificial Intelligence (BISFAI-89)*, 1989.

[9] R. Dechter. Enhancement schemes for constraint processing: backjumping, learning, and cutset decomposition. *Artificial Intelligence*, 41:273–312, 1990.

[10] R. Dechter and A. Dechter. Belief maintenance in dynamic constraint networks. In *Proceedings of the seventh national Conference of Artificial Intelligence (AAAI-88)*, pages 37–42, St. Paul, MN, 1988.

[11] R. Dechter and J. Pearl. Network-based heuristics for constraint satisfaction problems. *Artificial Intelligence*, 34:1–38, 1987.

[12] R. Dechter and J. Pearl. Tree clustering for constraint networks. *Artificial Intelligence*, pages 353–366, 1989.

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

[14] Y. El fattah and R. Dechter. Diagnosis for near tree structures. In *UCI technical report, 1994*, 1994.

[15] E. C. Freuder. A sufficient condition for backtrack-free search. *Journal of the ACM*, 29(1):24–32, 1982.

[16] H. Geffner and J. Pearl. An improved constraint propagation algorithm for diagnosis. In *Proceedings of IJCAI-87*, pages 1105–1111, Milan, Italy, 1987.

[17] M. R. Genesereth. The use of design descriptions in automated diagnosis. *Artificial Intelligence*, 24:411–436, 1984.

[18] J. Kohlas. Symbolic evidence, arguments and valuation networks. In *Uncertainty in Artificial Intelligence (UAI-93)*, Washington D.C, 1993.

[19] A. K. Mackworth and E. C. Freuder. The complexity of some polynomial network consistency algorithms for constraint satisfaction problems. *Artificial Intelligence*, 25(1), 1985.

[20] D. A. McAllester. An outlook on truth-maintenance. Technical Report AI Memo 551, MIT, Boston, Massachusetts, 1980.

[21] D. A. McAllester. Truth maintenance. In *Proceedings of AAAI-90*, pages 1109–1116, 1990.

[22] D. Mcdermott. A general framework for reason maintenance. *Artificial Intelligence*, 50(3):289–329, 1991.

[23] G. Provan. Complexity analysis of multiple-context TMSs in scene representation. In *Proceedings of AAAI-87*, pages 173–177, Seattle, Washington, 1987.

[24] L. G. Valiant. The complexity of enumeration and reliability problems. *Siam Jouranl of Computation*, 8(3):105–117, 1987.

[25] R. Dechter Z. Collin and S. Katz. On the feasibility of distributed constraint satisfaction. In *Proceedings of the twelfth International Conference of Artificial Intelligence (IJCAI-91)*, pages 318–324, Sidney, Australia, 1991.

Acknowledgement

We would like to thank Yousri El fattah for commenting on a recent version of this manuscript, and Dan Frost for helping with the figures. Preliminary versions of some of the contents of this paper appeared in [10] and in [8].

8 Appendix

Theorem 8 *: Algorithm* label generation *generates all and only minimal support sets.* □.

Proof: It is clear that the algorithm generates only t-supports. We will therefore focus on showing that any generated set is minimal and that any minimal t-support 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 $s = (X_1 = x_1, ..., X_t = x_t)$ be a generated t-support. We will show that s is a *minimal* t-support. Let h be the the furthest distance from Z of variables in s. For $h = 1$ it is known that only *minimal* t-supports are generated, i.e., the minimal child-supports. Assuming that the generated t-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 t-support. Let $s^P = ({X_1}^P = x_1, ..., {X_j}^P = x_j)$ be a subtuple of the t-support s having distance h such that all the participating variables have a common parent, P. Let $\bigcap_{x_i \in {X_i}^P} M^P_{{X_i}^P}(x_i) = A_P$. 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 t-support with lower distance. By performing this "reverse substitution" to all the variables in s having distance h we get a t-support 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* child-support of the corresponding label-dependent subset, (otherwise it would not be applied) the (nonreversed) substitution must result in a minimal t-support having distance h or less.

We will now show that if s is a minimal t-support 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: $\bigcap_{x_j \in X_j^{P_i}} M_{{X_j}^{P_i}{}^{P_i}}(x_j) = A_{P_i}$. 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 t-support of depth not greater than

$h-1$, 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 t-support 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 bottom up process. $\Box$.