Uncovering trees in constraint networks

Rina Dechter
Information & Computer Science
University of California, Irvine
Irvine, CA 92717

Itay Meiri
Cognitive Systems Laboratory
Computer Science Department
University of California
Los Angeles, CA 90024

Judea Pearl
Cognitive Systems Laboratory
Computer Science Department
University of California
Los Angeles, CA 90024

Abstract

This paper examines the possibility of removing redundant information from a given knowledge base and restructuring it in the form of a tree to enable efficient problem-solving routines. We offer a novel approach that guarantees removal of all redundancies that hide a tree structure. We develop a polynomial-time algorithm that, given an arbitrary binary constraint network, either extracts (by edge removal) a precise tree representation from the path-consistent version of the network or acknowledges that no such tree can be extracted. In the the latter case, a tree is generated that may serve as an approximation to the original network.

1 Introduction

Redundancy in constraint-based reasoning can be a mixed blessing. On one hand, redundant constraints can be used to explicate incompatible assignments that otherwise would be tried by a search algorithm. On the other hand, the presence of redundant constraints forces search algorithms to make unnecessary tests. The latter case is particularly aggravating when problems expressible in tree-structured networks are enriched with redundant constraints: When the tree structure is available, the problem can be solved in a backtrack-free manner, but if the tree is loaded with redundant information, the correct ordering of the search is obscured, which may lead to many deadends and to unnecessary consistency checks at each step.

The problem addressed in this paper is as follows. Given a binary constraint network, find whether it can be transformed into a tree-structured network without loss of information. If the answer is yes, find such a tree; if the answer is no, acknowledge failure.

The paper develops a polynomial-time algorithm that for a given binary constraint network, generates a tree T having the following characteristics. If any tree representation can be extracted by deleting edges (i.e., binary constraints) from the path-consistent version of the network, T represents the network exactly. However, if no tree representation can be extracted by such deletion, that fact is acknowledged. We show that tree extraction by edge-deletion is feasible only when the path-consistent network is minimal. Furthermore, when the given path-consistent network is minimal, we can issue a stronger guarantee: that if the tree T generated by our algorithm fails to represent the network, then no tree representation exists, even allowing for the introduction of edges that were absent in the original path-consistent network. In that case, T may serve as an approximation to the original network.

The algorithm works as follows. After enforcing path consistency, we examine all triplets of variables, identify the redundancies that exist in each triplet, and assign weights to the edges in accordance with the redundancies discovered. The tree generated, T, is a maximum-spanning-tree relative to these weights, and a (polynomial-time) test is then conducted to determine whether the tree represents the network precisely.

An added feature of the algorithm is that when the tree generated is recognized as an approximation, it can be further tightened by adding edges until a precise representation is achieved. This technique may be regarded as an alternative redundancy-removal scheme, complementing that proposed in [2], offering polynomial complexity and performance guarantees.

The general issue of removing redundancies has been investigated in the literature of relational databases [11, 3] and in the context of constraint networks [2]. The algorithm proposed here is related also to the problem of decomposing a relation [3], which will be discussed in detail in Section 8. While the method in [3] takes as input an explicit relation (i.e., the set of satisfying assignments), the input here consists of an unsolved constraint network.

2 Preliminaries and nomenclature

We first review the basic concepts of constraint satisfaction [4, 9].

A **network of binary constraints** consists of a set of variables $\{X_1, ..., X_n\}$ and a set of binary constraints on the variables. The **domain** of variable X_i, denoted by D_{X_i} or D_i, defines the set of values X_i may assume. A **binary constraint** R_{ij} on variables X_i and X_j, defined by $R_{i,j} \subseteq D_i \times D_j$, specifies the allowed pairs of values for X_i and X_j. If a pair (x, y) is allowed by the constraint R_{ij}, we denote $R_{ij}(x, y) = 1$; else, $R_{ij}(x, y) = 0$. Thus R_{ij} denotes a set of pairs, while $R_{ij}(x, y)$ is a predicate that is true iff $(x, y) \in R_{ij}$.

A binary constraint R_{ij} is **tighter**[1] than R'_{ij} (or conversely R'_{ij} is more **relaxed** than R_{ij}), denoted by $R_{ij} \subseteq R'_{ij}$, if every pair of values allowed by R_{ij} is also allowed by R'_{ij}. The most relaxed constraint is the **universal** constraint, which allows all pairs of the Cartesian product.

An assignment of a value to each variable that satisfies all the constraints is called a **solution**. The set of all solutions to network R constitutes a relation, denoted by $rel(R)$, whose attributes are the variables names. Formally, $rel(R) = \{x_i, ..., x_n | \forall\ i, j\ (x_i x_j) \in R_{ij}\}$. Two networks with the same variable set are **equivalent** iff they represent the same set of solutions.

A binary constraint network is associated with a **constraint graph**, where node i represents variable X_i, and an edge between nodes i and j represents a **direct constraint**, R_{ij}, between them, which is not the universal constraint. Other constraints are **induced** by paths connecting i and j. The constraint induced on i and j by a path of length m through nodes $i_0 = i,\ i_1, ..., i_m = j$, denoted by $R_{i_0, i_1, ..., i_m}$, represents the **composition** of the constraints along the path–namely, a pair of values $x \in D_{i_0}$ and $y \in D_{i_m}$ is allowed by the path constraint if there exists a sequence of values $v_1 \in D_{i_1}, v_{m-1} \in D_{i_{m-1}}$ such that $R_{i_0,i_1}(x, v_1),\ R_{i_1,i_2}(v_1, v_2)$, and $R_{i_{m-1},i_m}(v_{m-1}, y)$ are all evaluated to 1.

A network whose direct constraints are tighter than any of its induced path constraints is called **path consistent**. Formally, a path P of length m through nodes $i_0,\ i_1, ..., i_m$ is path consistent, iff $R_{i_0,i_m} \subseteq R_{i_0,i_1,...,i_m}$. Similarly, arc (i, j) is arc consistent if for any value $x \in D_i$,

[1] It should be "at least as tight as" but we use the shorter term "tighter" for convenience.

there exists a value $y \in D_j$ such that $R_{ij}(x, y)$. A network is arc and path consistent if all its arcs and paths are arc and path consistent, respectively. Any network can be converted into an equivalent arc- and path-consistent form in time $O(n^3)$[2] [10, 12]. The network resulting from applying arc- and path-consistency to R is denoted by $path(R)$.

Not every relation can be represented by a binary constraint network. The best network approximation of a given relation is called the **minimal network**; its constraints are the projections of the relation on all pairs of variables, namely, each pair of values allowed by the minimal network participates in at least one solution. Thus, the minimal network displays the tightest constraints between every pair of variables. Being a projection of the solution set, the minimal network is always arc and path consistent. Montanari [12] showed that the minimal network is unique. An equivalent definition of the minimal network is:

Definition 1 [12]. *A binary network R is* **minimal** *if for any network R' equivalent to R, R is tighter than R'.*

3 Problem statement

The problem addressed in this paper rests on the notions of **tree decomposition** and **tree reducibility**.

Definition 2 *A network R is* **tree decomposable** *if there exists a tree-structured network T on the same set of variables, such that R and T are equivalent (i.e., they represent the same relation). T is said to be a* **tree decomposition** *of R, and the relation ρ represented by R is said to be tree decomposable (by T). A path-consistent network R is* **tree reducible** *if it contains a tree-structured subnetwork T such that R is decomposable by T and, for all $(i, j) \in T$, $T_{ij} = R_{ij}$, the constraints in T are transferred from R with no alteration.*

The **tree-decomposability problem** for networks is defined as follows. Given a network R, decide whether R is tree decomposable. If the answer is positive, find a tree decomposition of R; else, acknowledge failure. The **tree-reducibility problem** is defined similarly: Given a network R, decide whether $path(R)$ is tree reducible. If the answer is positive, find a tree reduction of $path(R)$; else, acknowledge failure.

This paper provides a complete solution of the tree-reducibility problem and a partial solution to the tree-decomposability problem. We first show that if R is tree reducible, $path(R)$ is its minimal network. Subsequently, we provide an algorithm that finds a tree reduction if such exists, or acknowledges failure. In the latter case, we conclude either that a tree decomposition does not exist or, if a tree decomposition does exist, that the input network must be nonminimal.

Since the minimal network is known to be an effective representation, namely, a representation from which solutions can frequently (but not always) be extracted in linear time, the question is whether we gain very much by uncovering a tree representation from the minimal network. In the next three examples, we will demonstrate that recognizing a tree structure from the minimal network can sometimes save an exponential amount of computation and can always reduce computation by a factor of $O(n)$. First, we quote the following:

[2] Actually, the complexity is $O(n^3k^3)$, where k is the domain size; however, for simplicity, we assume the domain size is constant.

Lemma 1 [15]. *Given a minimal network R and a relation ρ, deciding whether $\rho = rel(R)$ is NP-hard.*

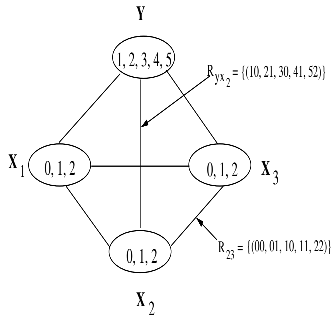
Thus, minimality in itself does not guarantee tractability of certain queries. Nonetheless, from the algorithm we present here, it follows that when the minimal network R has a tree representation, deciding whether $\rho = rel(R)$ is easy.

In the next two examples, we demonstrate that tree recognition may amount to exponential savings in search time even when the input network is minimal.

Example 1 *Consider a constraint network $R^{(n)}$ having $n+1$ variables $X_1, \ldots, X_n, Y$ with domains $X_1 = X_2 = \ldots = X_n = \{0, 1, 2\}$, $Y = \{1, 2, \ldots, n+1, n+2\}$. The constraints are:*

$$\begin{aligned} R_{X_i X_j} &= \{(0,0)(0,1)(1,0)(1,1)(2,2)\} \\ R_{Y X_i} &= \{(1,0)(2,0)\ldots(i-1,0)(i,1)(i+1,0)\ldots(n,0)(n+1,1)(n+2,2)\} \end{aligned}$$

The network for four variables Y, X_1, X_2, X_3 and its set of solutions is given in Figure 1.



$\rho =$

X_1	X_2	X_3	Y
1	0	0	1
0	1	0	2
0	0	1	3
1	1	1	4
2	2	2	5

Figure 1:

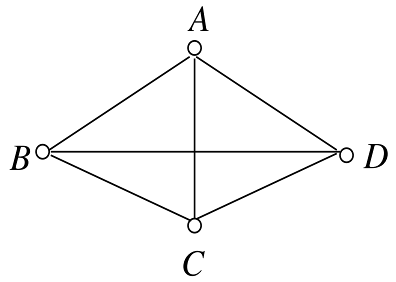
It is easy to see that $R^{(n)}$ is a minimal network. The network has a tree representation in which all the arcs connect to Y, hence any ordering that places Y as the first variable will lead to a backtrack-free search. Note, however, that, on the one hand, the number of constraints we need to test when extending a partial solution by one more value is $O(n)$, while, on the other, when eliminating the redundant arcs, only one constraint is tested at each step. More important, though, is the trashing we may encounter if we generate solutions in the wrong order (and there is not much to prevent us from selecting a "wrong" order of variables if we have no knowledge of the underlying structure of the network).

Assume that the order of the variables is $X_1, \ldots, X_n, Y$. In this case there are $2^n - n - 1$ tuples over $X_1, \ldots, X_n$ which are consistent relative to $X_1, \ldots, X_n$ (i.e., they satisfy all the constraints over those variables), and all of which are inconsistent with variable Y. In the worst case, our search space with this ordering of variables and with increasing ordering of value assignment yields $2^n - n - 1$ deadends. This grim picture could be avoided had we uncovered the tree that leads to the preferable variable ordering. The following two examples will be used to illustrate our algorithm.

Example 2 *Consider the network R having four variables A, B, C, D, with domains*

$$D_A = \{2,3\}, \quad D_B = \{2,3,4\}, \quad D_C = \{2,3,4\}, \quad D_D = \{2,6\}$$

The constraints are indicated explicitly in Figure 2.



$$\begin{aligned} R_{AB} &= \{(2,2)(2,4)(3,3)\} \\ R_{AC} &= \{(2,2)(2,4)(3,3)\} \\ R_{AD} &= \{(2,2)(2,6)(3,6)\} \\ R_{BC} &= \{(2,2)(2,4)(4,2)(4,4)(3,3)\} \\ R_{CD} = R_{BD} &= \{(2,2)(2,6)(4,2)(4,6)(3,6)\} \end{aligned}$$

The solution set is:

A	B	C	D
2	2	2	2
2	2	2	6
2	2	4	2
2	2	4	6
2	4	2	2
2	4	2	6
2	4	4	2
2	4	4	6
3	3	3	6

Figure 2: A binary network.

In any order of search, we will have to test all six constraints. This network is tree reducible. The constraints R_{BC}, R_{CD}, and R_{BD} are redundant and can be deleted. By recognizing this redundancy, we generate a representation that is much more effective; a consistent solution can be recovered by testing three constraints only. We may recognize now that the constraints between A and each of B, C, and D stand for the requirement that the value of A divides the values of B, C, and D respectively. This example can be scaled up to any number of variables, demonstrating again that even when there are no deadends, uncovering the tree may result in a reduction of constraint testing from $O(n^2)$ to $O(n)$.

Example 3 *Consider network R_2 whose variables A, B, C, D, E all have bi-valued domains* $\{0, 1\}$. The constraints are:

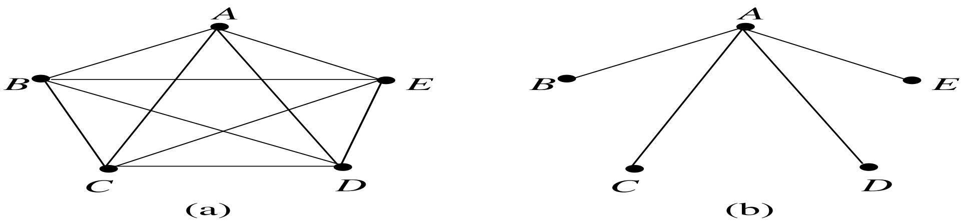


Figure 3: Constraint graphs for Example 2. Note that arc BC, DE in (a) denote universal constraints.

$$R_{AB} = R_{AC} = R_{BD} = R_{BE} = R_{CD} = R_{CE} = \{(0,1)(1,0)(1,1)\}$$

$$R_{AD} = R_{AE} = \{(0,0)(0,1)(1,1)\}$$

The constraint graph is given in Figure 3.

In this case, the tree $T = \{AB, AC, AD, AE\}$ is the only tree decomposition of this network.

The rest of the paper is organized as follows. Sections 4, 5, and 6 describe the tree-decomposition scheme. Section 7 provides some future extensions to general redundancy elimination and approximation, while Section 8 presents related work. Proofs of theorems can be found in the appendix.

4 Tree decomposition schemes

In this section, we present a solution to the tree-decomposition problem under the assumption that the starting network is minimal or becomes minimal by enforcing path-consistency. Subsequently, we show that this same algorithm solves the tree-reducibility problem in general. Tree decomposition comprises two subtasks: searching for a skeletal spanning tree, and determining the constraints on each edge of that tree. If the input network is minimal, the second subtask is superfluous because the constraints must be taken unaltered from the corresponding edges in the input network–namely, decomposability coincides with reducibility.

Lemma 2 *[12] Let T be a tree network. Then $path(T)$ is minimal.*

It follows that

Corollary 1 *If R is a path-consistent network that is not minimal, then R is not tree reducible*

Our problem can therefore be viewed as searching for a tree skeleton through the space of spanning trees. Since there are n^{n-2} spanning trees on n vertices (Cayley's Theorem [7]), a method more effective than exhaustive enumeration is required.

The notion of **redundancy** plays a central role in our decomposition schemes. Consider a consistent path $P = i_0,\ i_1, ..., i_m$. Recall that in the minimal network, the direct constraint R_{i_0,i_m} is tighter than the path constraint $R_{i_0,i_1,...,i_m}$. If the two constraints are identical, we say that edge (i_0, i_m) is **redundant** with respect to path P; it is also said to be redundant in the cycle C consisting of nodes $\{i_0,\ i_1, ..., i_m\}$. If the direct constraint is strictly tighter than the path constraint, we say that (i_0, i_m) is **nonredundant** with respect to P (or nonredundant in C). Another interpretation of redundancy is that any instantiation of the variables $\{i_0,\ i_1, ..., i_m\}$ which satisfies the constraints along P is allowed by the direct constraint R_{i_0,i_m}. Conversely, nonredundancy implies that there exists at least one instantiation that violates R_{i_0,i_m}.

Definition 3 *Let T be a tree, and let $e = (i, j) \notin T$. The unique shortest path in T connecting i and j, denoted by $P_T(e)$, is called the* **supporting path** *of e (relative to T). The cycle $C_T(e) = P_T(e) \cup \{e\}$ is called the* **supporting cycle** *of e (relative to T).*

Theorem 1 *Let $G = (V, E)$ be a minimal network. G is decomposable by a tree T iff every edge in $E - T$ is redundant in its supporting cycle.*

Theorem 1 gives a method of testing whether a network G is decomposable by a given tree T. The test takes $O(n^3)$ time, as there are $O(n^2)$ edges in $E - T$, and each redundancy test is $O(n)$.

Illustration: Consider Example 2. Tree $T_1 = \{AB, AC, AD\}$ is a tree decomposition, since edges BC, BD, and CD are redundant in triangles $\{A, B, C\}$, $\{A, B, D\}$, and $\{A, C, D\}$, respectively. Tree $T_2 = \{AD, BD, CD\}$ is not a tree decomposition, since edge AB is nonredundant in triangle $\{A, B, D\}$ (indeed, the tuple $(A = 2, B = 3, C = 3, D = 6)$ is a solution of T_2, but it is not a solution of the network).

An important observation about redundant edges is that they can be deleted from the network without affecting the set of solutions; the constraint specified by a redundant edge is already induced by other paths in the network. This seems to suggest the following decomposition scheme. Repeatedly select an edge redundant in some cycle C, delete it from the network, and continue until there are no cycles in the network or there are no redundant edges. Algorithm *brute-force decomposition (BFD)* is depicted in Figure 4.

Theorem 2 *Let G be a minimal network. Algorithm BFD produces a tree T iff G is tree decomposable by T.*

To prove Theorem 2, we must show that if the network is tree decomposable, *any* sequence of edge removals will generate a tree. A phenomenon that might prevent the algorithm

Algorithm (BFD)

1. $N \leftarrow E$;
2. while there are redundant edges in N, do
3. select an edge e that is redundant in some cycle C, and set $N \leftarrow N - \{e\}$
4. if N forms a tree, then G is decomposable by N
5. else, G is not tree decomposable;

Figure 4: BFD – A brute-force algorithm for tree-decomposition.

from reaching a tree structure is that of a **stiff cycle**, namely, one in which every edge is nonredundant (e.g., cycle $\{B, D, C, E\}$ in Example 3). It can be shown, however, that one of the edges in such a cycle must be redundant in another cycle when the network is tree decomposable.

The proof of Theorem 2 rests on the following three lemmas, which also form the theoretical basis For Section 5.

Lemma 3 *Let G be a path-consistent network, and let $e = (i_0, i_m)$ be an edge redundant in cycle $C = \{i_0, i_1, ..., i_m\}$. If $C' = \{i_0, i_1, ..., i_k, i_{k+l}, ..., i_m\}$ is an interior cycle created by chord (i_k, i_{k+l}), then e is redundant in C'.*

Lemma 4 *Let G be a minimal network decomposable by a tree T, and let $e \in T$ be a tree edge redundant in some cycle C. Then there exists an edge $e' \in C$, $e' \notin T$, such that e is redundant in the supporting cycle of e'.*

Lemma 5 *Let G be a minimal network decomposable by a tree T. If there exist $e \in T$ and $e' \notin T$ such that e is redundant in the supporting cycle of e', then G is decomposable by $T' = T - \{e\} \cup \{e'\}$.*

Corollary 2 *Let G be a path-consistent network. Algorithm BFD produces a tree T iff G is tree reducible.*

Algorithm BFD, although conceptually simple, is highly inefficient. The main drawback is that in Step 3 we might need to check redundancy against an exponential number of cycles. In the next section we show a polynomial algorithm that overcomes this difficulty by looking at cycles of length 3 (e.g., triangles) only. However, when redundancy is determined on triangles only (to bound complexity), the order by which such redundant edges are eliminated is important, as shown in Example 2. Edge AC is redundant in triangle ABC, and edge BD is redundant in ABD. However, if we remove both AC and BD, the resulting graph has no more triangles and we must stop. Alternatively, if we check redundancy on all triangles in advance, we realize that each of AC, BD, and BC is redundant in some triangle. However, eliminating all three constraints results in a network that does not represent the original relation, which has a tree decomposition. Guarding against misguided orderings of redundancy elimination is the essence of the algorithm given in the following section. We will provide a rank order of arcs such that redundancy elimination in that order is guaranteed to find a tree decomposition if such exists.

Algorithm TD
Input: A path-consistent network R
Output: A tree reduction of R, if one exists; else, R is not tree reducible.

1. w $<-$ a tree labeling of G;
2. T $<-$ $MWST$ of G w.r.t. w;
3. test whether G is decomposable by T;
4. if the test fails, G is not tree decomposable; else, return the tree T.

Figure 5: TD – A family of tree-decomposition algorithms.

5 Tree, triangle, and redundancy labelings

In this section, we present a new tree-decomposition scheme (which can be regarded as an efficient version of BFD) whereby the criterion for removing an edge is essentially precomputed. To guide BFD in selecting redundant edges, we first impose an ordering on the edges such that nonredundant edges will always attain a higher ranking than redundant ones. Given such an ordering, we could either remove edges of low rank, or apply the dual method and construct a tree containing the preferred edges by finding a *maximum weight spanning tree (MWST)* relative to the given ordering. We focus here on the second method.

We define three types of labelings of edges: tree labeling, redundancy labeling, and triangle labeling. A labeling is a **tree labeling** iff the $MWST$ algorithm produces a tree decomposition when one exists. **A redundancy labeling** is a labeling satisfying some condition of redundancy in cycles. We show that the existence of redundancy labeling is necessary and sufficient for the existence of tree labeling, and hence for tree decomposition. Finally, a **triangle labeling** is one that captures redundancy in triangles only. We show that triangle labeling implies redundancy labeling, and hence a tree decomposition.

Definition 4 *Let $G = (V, E)$ be a minimal network. A* **labeling** *w of G is an assignment of weights to the edges, where the weight of edge $e \in E$ is denoted by $w(e)$. w is said to be a* **tree labeling** *if it satisfies the following condition. If G is tree decomposable, then G is decomposable by tree T iff T is an MWST of G with respect to w.*

Finding a tree labeling essentially solves the tree-decomposability problem, simply by following the steps of algorithm *tree decomposition* (TD) shown in Figure 5. TD stands for a family of algorithms, where each algorithm is driven by a different labeling w. Steps 2-4 can be implemented in $O(n^3)$: Step 2 can use any $MWST$ algorithm, such as the one by Prim, that is $O(n^2)$ (see [7]); Steps 3-4, deciding whether G is decomposable by T, are $O(n^3)$, as explained in Section 4 (Theorem 1).

We now turn our attention to Step 1, namely, computing a tree labeling. This will be done in two steps. We first introduce a necessary and sufficient condition for a labeling to qualify as a tree labeling, and then synthesize an $O(n^3)$ algorithm that returns a labeling w satisfying this condition. As a result, with this labeling the total running time of TD is bounded by $O(n^3)$.

Definition 5 *Let $G = (V, E)$ be a minimal network. A labeling w of G is called a* **redundancy labeling** *if it satisfies the following condition. For any tree T and any two edges*

$e' \in E-T$ and $e \in T$ such that e is on the supporting cycle $C_T(e')$ of e', if G is decomposable by T, then

$$(i) \quad w(e') \leq w(e) \tag{1}$$

$$(ii)\ e \text{ is redundant in } C_T(e') \text{ whenever } w(e') = w(e) \tag{2}$$

Lemma 6 *Let w be any labeling of a minimal network G. w is a tree labeling iff w is a redundancy labeling.*

Having established this equivalence, the next step is to construct a labeling that satisfies conditions (1) and (2).

Definition 6 *A labeling w of network G is a* **triangle labeling** *if for any triangle $t = \{e_1, e_2, e_3\}$ the following conditions are satisfied.*
(i) If e_1 is redundant in t, then

$$w(e_1) \leq w(e_2), \ \ w(e_1) \leq w(e_3) \tag{3}$$

(ii) If e_1 is redundant in t and e_2 is nonredundant in t, then

$$w(e_1) < w(e_2) \tag{4}$$

Conditions (3) and (4) will be called **triangle constraints**.
Illustration: Consider the minimal network of Example 3. Analyzing redundancies relative to all triangles leads to the triangle constraints depicted in Figure 6. Each node in the figure represents an edge of the minimal network, and arc $e_1 \rightarrow e_2$ represents the triangle constraint $w(e_1) < w(e_2)$ (for clarity, all arcs from bottom layer to top layer were omitted). It so happens that only strict inequalities were imposed in this example. A triangle labeling w can be constructed easily by assigning the following weights:

$$w(AB) = w(AC) = w(AD) = w(AE) = 3$$

$$w(BD) = w(BE) = w(CD) = w(CE) = 2$$

$$w(BC) = w(DE) = 1$$

Note that the tree $T = \{AB, AC, AD, AE\}$, which decomposes the network, is an $MWST$ relative to these weights, a property that we will show to hold in general.

Clearly, conditions (3) and (4) are easy to verify as they involve only tests on triangles. In Lemma 8, we will indeed show that they are sufficient to constitute a redundancy labeling, hence a tree labeling. Moreover, a labeling satisfying conditions (3) and (4) is easy to create primarily because, by the following Lemma 7, such a labeling is guaranteed to exist for any path-consistent (hence for any minimal) network. Note that this is by no means obvious, because there might be two sets of triangles imposing two conflicting constraints on a pair (a, b) of edges: one requiring $w(a) \leq w(b)$, and the other $w(a) > w(b)$.

Lemma 7 *Any path-consistent network admits a triangle labeling.*

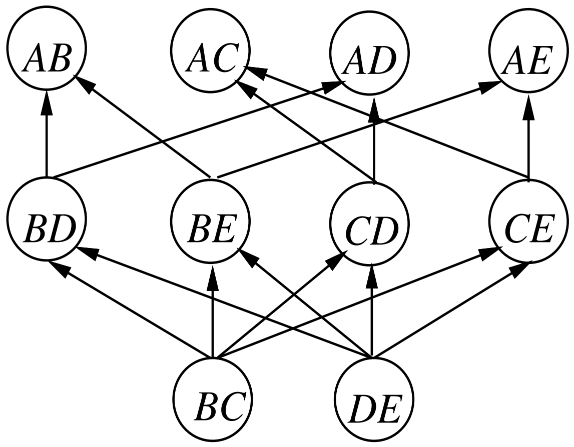


Figure 6: Triangle constraints for Example 2.

Notice that when there is no redundancy, any labeling is a triangle labeling. The idea behind triangle labelings is that all redundancy information necessary for tree decomposition can be extracted from individual triangles rather then cycles. By Lemma 3, if an edge is redundant in a cycle, it must be redundant in some triangle. Contrapositively, if an edge is nonredundant in all triangles, it cannot be redundant in any cycle, and thus must be included in any tree decomposition. To construct a tree decomposition, we must, therefore, include all those necessary edges (note that they attain the highest ranking) and then proceed by preferring edges that are nonredundant relative to others. The correctness of the next lemma rests on these considerations.

Lemma 8 *Let G be a minimal network. If w is a triangle labeling of G, then it is also a redundancy labeling.*

We can conclude:

Theorem 3 *Let G be a minimal network, and assume TD uses a triangle labeling w of G. G is tree decomposable iff TD finds a tree decomposition of G.*

Theorem 4 *Let G be a path-consistent network, and assume TD uses a triangle labeling w of G. G is tree reducible iff TD finds a tree reduction of G.*

From here on we will assume that the labeling w computed by TD in Step 1 is a triangle labeling. What remains to be shown is that given any minimal network $G = (V, E)$, a triangle labeling can be formed in $O(n^3)$ time. Algorithm *triangle labeling* (TL), shown in Figure 7, accomplishes this task.

Let us consider algorithm TL in detail. First, it constructs a graph, G_1, that displays the triangle constraints. Each node in G_1 represents an edge of G, and arc $u \rightarrow v$ stands for a triangle constraint $w(u) \leq w(v)$ or $w(u) < w(v)$. The construction of G_1 (Steps 1-3) takes $O(n^3)$ time, since there are $O(n^3)$ triangles in G, and the time spent for each triangle is constant.

Algorithm *TL*
Input: An arc- and path-consistent network R
Output: A triangle labeling w.

1. create directed graph $G_1 = (V_1, E_1)$ with $V_1 = E$ and $E_1 = \Phi$;
2. for each triangle $t = \{e_i, e_j, e_k\}$ in G, do
   if edge e_i is redundant in t, then add arcs $e_i \rightarrow e_j$ and $e_i \rightarrow e_k$ to G_1;
3. set $G_2 = (V_2, E_2)$ as the superstructure of G_1; $V_2 = \{C_1, ..., C_e\}$.
4. compute a topological ordering w for V_2;
5. for $i := 1$ to $|V_2|$, do
6. for each edge e in C_i, do
   $w(e) \leftarrow w(C_i)$;

Figure 7: TL - An algorithm for constructing a triangle labeling.

Consider a pair of nodes, u and v, in G_1. It can be verified that if the nodes belong to the same strongly connected component (i.e., they lie on a common directed cycle),[3] their weights must satisfy $w(u) = w(v)$. If they belong to two distinct components but there exists a directed path from u to v, their weights must satisfy $w(u) < w(v)$. These relationships can be effectively encoded in the *superstructure* of G_1 [7]. Informally, the superstructure is formed by collapsing all nodes of the same strongly connected component into one node, while keeping only arcs that go across components. Formally, let $G_2 = (V_2, E_2)$ be the superstructure of G_1. Node $C_i \in G_2$ represents a strongly connected component, and a directed arc $C_i \rightarrow C_j$ implies that there exists an edge $u \rightarrow v$ in G_1, where $u \in C_i$ and $v \in C_j$. Identifying the strongly connected components, and consequently constructing the superstructure (Step 4), takes $O(n^3)$ (a time proportional to the number of edges in G_1 [7]).

It is well known that the superstructure forms a directed acyclic graph (DAG), and, moreover, that the nodes of the DAG can be topologically ordered, namely, they can be given distinct weights w such that if there exists an arc $i \rightarrow j$, then $w(i) < w(j)$. This can be accomplished (Step 4) in time proportional to the number of edges, namely, $O(n^3)$. Finally, recall that each node in G_2 stands for a strongly connected component, C_i, in G_1, which in turn represents a set of edges in G. If we assign weight $w(C_i)$ to these edges, w will comply with the triangle constraints, and thus will constitute a triangle labeling. Since all steps are $O(n^3)$, the entire algorithm is $O(n^3)$.

These considerations are summarized in the following theorem.

Theorem 5 *Given a path-consistent network R, algorithm TL generates a triangle labeling of R in $O(n^3)$ steps.*

Corollary 3 *The tree decomposability of a minimal network G can be decided in $O(n^3)$ steps. Furthermore, if it exists, a tree decomposition of G can be generated in $O(n^3)$.*

[3] A strongly connected component of a directed graph is a maximal set of node U such that for every pair A and B in U, there is a directed cycle containing A and B.

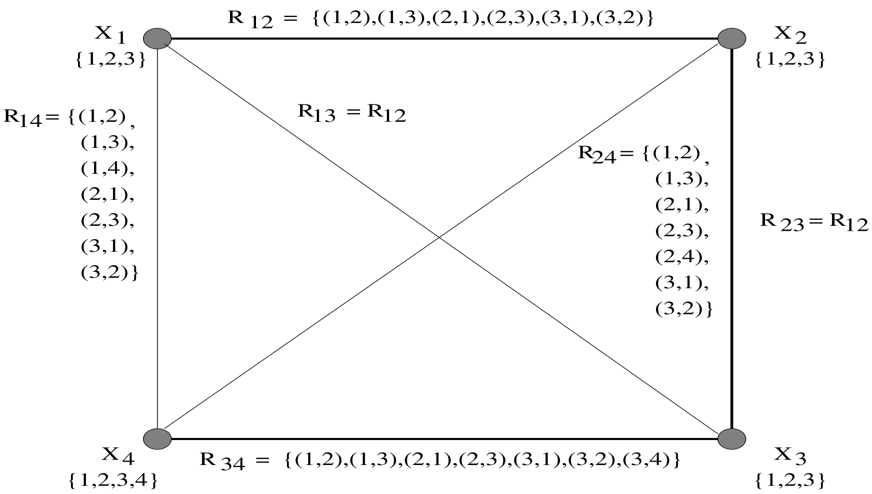


Figure 8: A nonminimal path-consistent network representing tree-decomposable relation
.

6 Tree decomposition vs. tree reduction

Given an arbitrary network R (not necessarily minimal), we wish to determine whether R is tree decomposable. If it were the case that any tree-decomposable network becomes minimal by enforcing path-consistency, then algorithm TD preceded by path-consistency would solve tree decomposability for the general case. This is not true, however. There are path-consistent networks that are not minimal and yet are tree decomposable.

Example 4 [13]. *Consider the following network (see Figure 8) on four variables.*

$$X = \{X_1, X_2, X_3, X_4\}, \quad D_i = \{1,2,3\}, i = 1,2,3, \quad D_4 = \{1,2,3,4\}$$

$$\begin{aligned}
R_{12} &= \{(1,2),(1,3),(2,1),(2,3),(3,1),(3,2)\} \\
R_{13} &= \{(1,2),(1,3),(2,1),(2,3),(3,1),(3,2)\} \\
R_{14} &= \{(1,2),(1,3),(1,4),(2,1),(2,3),(3,1),(3,2)\} \\
R_{23} &= \{(1,2),(1,3),(2,1),(2,3),(3,1),(3,2)\} \\
R_{24} &= \{(1,2),(1,3),(2,1),(2,3),(2,4),(3,1),(3,2)\} \\
R_{34} &= \{(1,2),(1,3),(2,1),(2,3),(3,1),(3,2),(3,4)\}
\end{aligned}$$

which represents a tree-decomposable relation having one solution, $\rho = \{(X_1 = 1, X_2 = 2, X_3 = 3, X_4 = 4)\}$. *The network is path consistent but not minimal.*

If we try to apply our algorithm to this network, we will find no redundancy in any triangle or in any cycle. Indeed, the algorithm assumes that the given network is minimal, and it will not be able to recognize such redundancy, which is hidden in its minimal network.

As noted at the outset, our tree-reduction algorithm, TD, will decide tree decomposition whenever path consistency produces the minimal network. Theorem 3 leads to the following observation.

Theorem 6 *Algorithm TD decides tree decomposition for the following classes of networks:*

1. *Tree-reducible networks*
2. *Path-consistent row-convex networks*
3. *Implicational constraints networks*
4. *Binary* $(0, 1)$ *networks*
5. *Distributive networks*

Row-convex networks involve constraint matrices having consecutive sequences of 1's [16]. Distributive networks employ relations for which the composition operation is distributive over intersection [12]. Implicational constraints networks are binary networks where, in each constraint, every value can match none, one, or all of the values of the other variable [8, 1]. Implicational constraints networks are a special case of row-convex networks

that are closed under path-consistency. Since the path-consistent networks listed in 2-5 are all globally consistent, the saving that is introduced by redundancy elimination is at most $O(n)$ (i.e., testing one constraint rather then $O(n)$ at each stage).

7 Redundancy elimination and approximation

Another application of TD is redundancy removal. Given a network R (not necessarily tree decomposable), it is sometimes desirable to remove as many redundant edges as possible. Our scheme provides an effective removal heuristic, alternative to that of [2]. In [2], an algorithm called *path-redundancy* is introduced. It eliminates, in some sequence, edges that are path redundant relative to a set of paths. If we apply algorithm TD first, we can use its weights to guide the order of path-redundancy elimination, thus guaranteeing that a tree will be identified if one exists. Alternatively, we can first apply TD and then, if the tree generated does not represent the network precisely, add nonredundant edges until a precise representation obtains.

TD can also be used for approximation: Given a network R, find a tree network that constitutes a good approximation of R. The tree T generated by TD provides an upper bound of R, as it enforces only a subset of the constraints. The quality of this approximation should therefore be evaluated in terms of the tightness, or specificity, of T.

Conjecture 1 *If R is a minimal network, the tree T generated by TD is* **most specific** *in the following sense: no other tree T', extracted from the network, satisfies $rel(T') \subset rel(T)$.*

Although we have not yet found a proof, the conjecture has managed to endure all attempts to construct a counterexample.

8 Related work: Decomposing a relation

The problem of tree decomposition was solved for general relations. Given a relation ρ, the problem is to determine whether ρ is tree decomposable. We first describe how TD can be employed to solve this problem, and then compare it with the solution presented in [3].

We start by generating the minimal network M from ρ. We do this by projecting ρ on each pair of variables. We then apply TD to solve tree decomposability for M. If M is not tree decomposable, ρ cannot be tree decomposable, because then, there would be a tree T satisfying $\rho = rel(T) \subset rel(M)$, violating the minimality of M [12]. If M is decomposable by the generated tree T, we still need to test whether $rel(T) = \rho$ (note that M may not represent ρ precisely). This can be done by comparing the sizes of the two relations: ρ is decomposable by T iff $|\rho| = |rel(T)|$. Generating M takes $O(n^2 \mid \rho|)$ operations, while $|rel(T)|$ can be computed in $O(n)$ time [5]; thus, the total time of this method is $O(n^2 \mid \rho|)$.

An alternative solution to the problem was presented in [3]. It computes for each edge a numerical measure, w, based on the frequency with which each pair of values appears in the relation. First, the following parameters are computed:

$n(X_i = x_i)$ = number of tuples in ρ in which variable X_i attains value x_i

$n(X_i = x_i,\ X_j = x_j)$ = number of tuples in ρ in which both $X_i = x_i$ and $X_j = x_j$

Then, each edge $e = (i, j)$ is assigned the weight

$$w(e) \sum_{x_i, x_j \in X_i, X_j} n(x_i, x_j) log \frac{n(x_i, x_j)}{n(x_i)n(x_j)} \tag{5}$$

It has been shown that this labeling, w, is indeed a tree labeling, also requiring $O(n^2 \mid \rho|)$ computational steps.

Of the two schemes, the method presented in this paper has three advantages. First, it does not need the precision required by the log function. Second, it offers a somewhat more effective solution in cases where ρ is not available in advance but is observed incrementally through a stream of randomly arriving tuples. Finally, it is conceptually more appealing, since the removal of each edge is meaningfully justified in terms of being redundant.

9 Conclusion

The problem addressed in this paper is best viewed as a task of "knowledge compilation" [14, 6], in which knowledge specified in one form is compiled into a more manageable form, so as to accommodate a given stream of queries. The compilation task treated in this paper concerns the decomposition of a constraint network into a tree – a structure known to facilitate tractable answers to a wide spectrum of queries.

This paper develops a tractable decomposition scheme that requires $O(n^3)$ time and solves the problem for minimal networks and for any path-consistent network from which a tree decomposition can be extracted by deleting edges. The technique is complete for several classes of networks for which path-consistency produces the minimal network. Row-convex and distributive networks are two such classes.

The theoretical contribution of this paper lies in delineating the extent to which one can generate trees and remove redundancies by examining only triplets of variables. That such local examination could be sufficient for certain classes of networks is an intruiging finding, and should add to our general understanding of dependency and redundancy in constraint networks.

We can only speculate about the applicability of this method for large, real-life problems. The method can certainly be useful for guiding removal of redundancies and for generating tree networks that provide upper-bound approximations. However, the prospects for uncovering tree structures in real-life databases, while a serious possibility in highly structured domains (i.e., temporally indexed relationships), may be rather dim; we suspect that, in practice, most networks will not be tree decomposable. In such cases, the effectiveness of our technique would rest upon the goodness of the approximation provided by the tree generated and on how well the redundancies discovered are exploited.

Acknowledgments

We thank Irina Rish for Example 4 which resolved a longstanding conjecture. This work was supported in part by Air Force Grant #AFOSR 90 0136, National Science Foundation Grant #IRI 9200918, National Science Foundation Grant #IRI 9157636, Toshiba of America, and a grant from Xerox.

References

[1] Cooper, M.C., D.A. Cohen, and P.G. Jeavons, "Characterizing Tractable Constraints," *Artificial Intelligence* 65, 347-361, 1994.

[2] Dechter, A., and R. Dechter, "Removing Redundancies in Constraint Networks," in *Proceedings of AAAI-87*, Seattle, WA, 105-109, 1987.

[3] Dechter, R., "Decomposing a Relation into a Tree of Binary Relations," *Journal of Computer and System Sciences,* 41, 2-24, 1990.

[4] Dechter, R., "Constraint Networks," in *Encyclopedia of Artificial Intelligence*, 2nd edition, Wiley and Sons, 276-285, 1992.

[5] Dechter, R., and J. Pearl, "Network-Based Heuristics for Constraint Satisfaction Problems," *Artificial Intelligence,* 34(1), 1-38, 1987.

[6] Dechter, R., and J. Pearl, "Structure Identification in Relational Data," *Artificial Intelligence*, 58, 237-270, 1992.

[7] Even, S., *Graph Algorithms,* Computer Science Press, 1979.

[8] Kirousis, L.M., "Fast Parallel Constraint Satisfaction," *Artificial Intelligence*, 64, 147-160, 1993.

[9] Mackworth, A.K., "Constraint Satisfaction," in *Encyclopedia of Artificial Intelligence*, 2nd edition, Wiley and Sons, 276-285, 1992.

[10] Mackworth, A.K., and E.C. Freuder, "The Complexity of Some Polynomial Network Consistency Algorithms for Constraint Satisfaction Problems," *Artificial Intelligence,* 25(1), 65-74, 1985.

[11] Maier, D., *The Theory of Relational Databases*, Computer Science Press, 1983.

[12] Montanari, U., "Networks of Constraints: Fundamental Properties and Applications to Picture Processing," *Information Sciences,* 7, 95-132, 1974.

[13] Rish, I., Personal communication, April 1995.

[14] Selman, B., and H.A. Kautz, "Tractability through Theory Approximation," AI Technical Report, AT&T Bell Laboratories, Murray Hill, NJ, 1992.

[15] Ullman, J., Personal communication.

[16] Van Beek, P. "On the Minimality and the Decomposability of Constraint Networks," *in Proceedings of AAAI-92*, San Jose, CA, 447-452, 1992.

Appendix: Proofs of Theorems

Theorem 1 Let $G = (V, E)$ be a minimal network. G is decomposable by a tree T iff every edge in $E - T$ is redundant in its supporting cycle.

Proof: Assume G is decomposable by T. Suppose there is an edge $(i, j) \in E - T$ that is nonredundant relative to its supporting path P_{ij}. Thus, there exists an instantiation of the variables on P_{ij} which satisfies the constraints along P_{ij}, but the pair of values (x, y), assigned to variables i and j, is disallowed by R_{ij}. Since the network is arc consistent, this instantiation can be extended to a complete solution of T. However, since the pair (x, y) is disallowed by R_{ij}, T is not equivalent to G, and thus cannot be a tree decomposition; contradiction.

The other direction is rather obvious. If any edge in $E - T$ is redundant in its supporting cycle, it can be deleted from the network without affecting the set of solutions. Thus, T is equivalent to G, and it is a tree decomposition. $\Box$

Lemma 2 Let T be a tree network, then $path(T)$ is minimal.
Proof: The reason is that any pair of values allowed by a unique path of tree edges can be extended to a full solution and therefore will appear in the minimal network. $\Box$
Lemma 3 Let G be a path-consistent network, and let $e = (i_0, i_m)$ be an edge redundant in cycle $C = \{i_0, i_1, ..., i_m\}$. If $C' = \{i_0, i_1, ..., i_k, i_{k+l}, ..., i_m\}$ is an interior cycle created by chord (i_k, i_{k+l}), then e is redundant in C'.

Proof: From path-consistency, we have

$$R_{i_k, i_{k+l}} \subseteq R_{i_k, i_{k+1}, ..., i_{k+l}} \tag{6}$$

Composition of constraints preserves tightness, thus

$$R_{i_0, ..., i_k, i_{k+l}, ..., i_m} \subseteq R_{i_0, ..., i_k, i_{k+1}, ..., i_{k+l}, ..., i_m} \tag{7}$$

Since (i_0, i_m) is redundant in C, we have

$$R_{i_0, ..., i_k, i_{k+1}, ..., i_{k+l}, ..., i_m} \subseteq R_{i_0, i_m} \tag{8}$$

From (7) and (8), we obtain

$$R_{i_0, ..., i_k, i_{k+l}, ..., i_m} \subseteq R_{i_0, i_m} \tag{9}$$

From path-consistency, $R_{i_0, i_m} \subseteq R_{i_0, ..., i_k, i_{k+l}, ..., i_m}$, and thus (i_0, i_m) is redundant in C'. $\Box$

Lemma 4 Let G be a minimal network decomposable by a tree T, and let $e \in T$ be a tree edge redundant in some cycle C. Then there exists an edge $e' \in C, e' \notin T$, such that e is redundant in the supporting cycle of e'.

Proof: Assume that the vertices along C are $v_1, ..., v_m$, where $e = (v_1, v_m)$. Without loss of generality, we may assume that v_1 is not a leaf in T (otherwise, reverse the order of the vertices along C). Let k be the highest index such that there exists a path $P_{1,k}$ in T from v_1 to v_k not passing through v_m. Note that $k > 1$ since v_1 is not a leaf.

Consider the path $P = P_{1,k} \cup \{e\}$ which is entirely contained in T. There exists a path in T connecting vertex v_{k+1} to a unique vertex, v, on P. Clearly $v = v_m$; otherwise, there would be a path in T from v_1 to v_{k+1} not passing through v_m, violating the assumption that v_k is the highest such vertex. Therefore, there exists a path in T from v_{k+1} to v_m. Let $P_{k+1,m}$ denote this path.

Let $e' = (v_k, v_{k+1})$. The supporting cycle of e' is

$$C_T(e') = P_{1,k} \cup \{(v_k, v_{k+1})\} \cup P_{k+1,m} \cup \{e\} \tag{10}$$

To complete the proof we now show that e is redundant in $C_T(e')$. From Lemma 3, since e is redundant in C, it is also redundant in the quadrangle $\{v_1, v_k, v_{k+1}, v_m\}$. However, (v_1, v_k) and (v_{k+1}, v_m) are redundant with respect to their supporting paths, $P_{1,k}$ and $P_{k+1,m}$, respectively. Thus, e is redundant in $C_T(e')$. □

Lemma 5 Let G be a minimal network decomposable by a tree T. If there exist $e \in T$ and $e' \notin T$ such that e is redundant in the supporting cycle of e', then G is decomposable by $T' = T - \{e\} \cup \{e'\}$.

Proof: By Theorem 1, we need to show that every edge is redundant with respect to its supporting path relative to T'. Let (i, j) be any edge in $E - T'$, and let P be its supporting path in T'. Consider an instantiation of the variables on P which satisfies the constraints along P. Let x and y be the values assigned to i and j, respectively, by this instantiation. We will show that they are also allowed by the direct constraint $R_{i,j}$.

Since the network is arc consistent, we can extend this partial instantiation to include the rest of the variables, in accordance with the constraints of T'. Since e is redundant in its supporting cycle in T' (it is redundant in $C_T(e') = C_{T'}(e)$), the instantiation satisfies the direct constraint represented by e. Thus, since $T \subseteq T' \cup \{e\}$, the instantiation satisfies all the constraints of T. Since T is a tree decomposition, the pair (x, y) is allowed by $R_{i,j}$. □

Theorem 2 Let G be a minimal network. Algorithm BFD produces a tree T iff G is decomposable by T.

Proof: Clearly, if BFD produces a tree, it constitutes a tree decomposition. Conversely, we will show that if the network is tree decomposable, BFD produces a tree decomposition.

We claim that during the execution of BFD, the following invariant is maintained: there exists a tree decomposition T such that $T \subseteq N$.

Initially the invariant holds, since the network is decomposable by some tree $T \subset E = N$. Now assume that the invariant holds before edge e is deleted from N. e is deleted because it is redundant in some cycle C. If $e \notin T$, then the invariant trivially holds after the deletion of e. If $e \in T$, then, according to Lemma 4, there exists an edge $e' \notin T$ such that e is redundant in its supporting cycle. Then, from Lemma 5, $T' = T - \{e\} \cup \{e'\}$ is a tree decomposition of G, and $T' \subseteq N$. Hence, the invariant holds after e is deleted.

To complete the proof, we need to show that upon termination, N constitutes a tree. Suppose N contains a cycle C. Since N always contains a tree decomposition T, there is an edge $e \in C$ which is redundant in its supporting cycle, and thus can be deleted. Therefore, when BFD terminates, N forms a tree. $\Box$

Lemma 6 Let w be a labeling of a minimal network G. w is a tree labeling iff w is a redundancy labeling.

Proof: If G is not tree decomposable, the theorem trivially holds. Now assume G is tree decomposable. We use a well-known fact from graph theory, called the *MWST property*, which says that a tree T is an $MWST$ iff every non-tree edge is an edge of minimum weight in its supporting cycle.

if part: Let w be a redundancy labeling of G. We shall show that w is also a tree labeling, namely, for any tree $T \subseteq E$, G is decomposable by T iff T is an $MWST$ with respect to w.

Let $T \subseteq E$ be a tree decomposition of G. From condition (1) and the MWST property, we conclude that T is an $MWST$ with respect to w.

Conversely, let T be an $MWST$ with respect to w. We show that if G is decomposable by a tree T', then it is also decomposable by T. The proof is by induction on $k = |T' - T|$, namely, the number of edges contained in T' but not in T.

Clearly, for $k = 0$, G is decomposable by $T = T'$. Now assume that if G is decomposable by T', such that $|T' - T| = k$, then it is also decomposable by T. We have to show that if G is decomposable by tree T', such that $|T' - T| = k + 1$, then it is also decomposable by T.

Let T' be a tree decomposition, where $|T' - T| = k + 1$. Let e be an edge in $T - T'$. Clearly, in $C_{T'}(e)$, its supporting cycle relative to T', there are edges of $T' - T$; let E' denote this set of edges. We first show that there exists an edge $e' \in E'$ such that $w(e') \leq w(e)$.

Consider $T - \{e\}$. Deleting e from T divides T into two subtrees T_1 and T_2. At least one of the edges in E' connects a vertex in T_1 with a vertex in T_2; let e' denote such an edge. We observe that e is in the supporting cycle of e' relative to T. Then, by applying the $MWST$ property to T, $w(e') \leq w(e)$.

Consider again $C_{T'}(e)$. From condition (1), $w(e) \leq w(e')$, hence $w(e) = w(e')$. From condition (2), we conclude that e' is redundant in $C_{T'}(e)$. By Lemma 5, $T'' = T' - \{e'\} \cup \{e\}$ is a tree decomposition of G. Furthermore, $|T'' - T| = k$. Thus, by the induction hypothesis, G is decomposable by T.

only if part: Let w be a tree labeling of G. We shall show that w is a redundancy labeling.

Suppose w is not a redundancy labeling. Then there exists a tree decomposition of G, $T \subseteq E$, and a non-tree edge e', having a supporting cycle $C_T(e')$, for which either condition (1) or condition (2) is violated. There are two cases, depending on which condition is violated.

Case 1: If condition (1) is violated, then there exists a tree edge $e \in C_T(e')$ such that $w(e) < w(e')$. By the $MWST$ property, T is not an $MWST$ relative to w. However, G is decomposable by T, and hence w is not a tree labeling; contradiction.

Case 2: If condition (2) is violated, then there exists a tree edge $e \in C_T(e')$ such that $w(e) = w(e')$ but e is nonredundant in $C_T(e')$. Clearly, $T' = T - \{e\} \cup \{e'\}$ is an $MWST$ relative to w. However, T' is not a tree decomposition, since e is nonredundant in $C_{T'}(e) = C_T(e')$, its supporting cycle in T'. Thus, w is not a tree labeling; contradiction. $\Box$

Lemma 7 Any path-consistent network admits a triangle labeling.

Proof: Suppose not. Therefore, there are two conflicting constraints, namely, there is a pair of edges $e', e'' \in E$ for which one set of triangle constraints requires $w(e') > w(e'')$, whereas another set of triangle constraints requires $w(e') \leq w(e'')$. Together, there exists a sequence of edges $e\prime = e_1, e_2, ..., e_k = e'', ..., e_m = e'$ for which the triangle constraints require

$$w(e_1) \leq ... \leq w(e_k) \leq ... \leq w(e_l) < w(e_{l+1}) \leq w(e_{l+2}) \leq ... \leq w(e_m) \tag{11}$$

Without loss of generality we can rename the edges, and the constraints may be written as

$$w(e_1) \leq ... \leq w(e_{m-1}) \leq w(e_m) < w(e_{m+1}) \tag{12}$$

where $e_{m+1} = e_1$, and the strict inequality is last. Let $t_2, ..., t_m, t_{m+1}$ be the corresponding sequence of triangles, namely, t_i contains edges e_{i-1} and e_i for $i = 2, ..., m+1$.

We now show by induction that for all $i, 2 \leq i \leq m$, there exists a cycle C_i containing e_1 and e_i, in which e_1 is redundant.

For $i = 2$, triangle t_2 contains e_1 and e_2, and imposes the constraint $w(e_1) \leq w(e_2)$. Hence, e_1 is redundant in $C_2 = t_2$.

Now assume that there exists a cycle C_i containing e_1 and e_i, in which e_1 is redundant. Consider triangle t_{i+1}. It contains both e_i and e_{i+1}, and, from the triangle constraint, e_i is redundant in t_{i+1}. Let v_1, v_2, and v_3 be the vertices of t_{i+1}, where $e_i = (v_1, v_2)$. Clearly, vertices v_1 and v_2 lie on C_i. There are two cases depending on the location of v_3.

Case 1: v_3 is not in C_i. Let the third edge of t_{i+1} (besides e_i and e_{i+1}) be c_{i+1}, and let $C_{i+1} = C_i - \{e_i\} \cup \{e_{i+1}, c_{i+1}\}$. Clearly, e_1 is redundant in C_{i+1}.

Case 2: v_3 is in C_i. Therefore, e_{i+1} is a chord of C_i, and it divides C_i into two interior cycles, C_{i_1} that contains e_1 and e_{i+1}, and C_{i_2}. By Lemma 3, since e_1 is redundant in C_i, it is also redundant in $C_{i+1} = C_{i_1}$.

We have now proved that there exists a cycle containing e_1 and e_m in which e_1 is redundant. However, e_1 and e_m are adjacent (they are both contained in triangle t_{m+1}). Therefore, from Lemma 3, e_1 is redundant in t_{m+1}. Yet, triangle t_{m+1} imposes the constraint $w(e_m) < w(e_1)$, implying that e_1 is nonredundant in t_{m+1}; contradiction. □

Lemma 8 Let G be a minimal network. If w is a triangle labeling of G, then it is also a redundancy labeling.

Proof: If G is not tree decomposable, the theorem trivially holds. Now assume G is decomposable by tree T. Let $e' \notin T$ and $e \in T$ be edges such that e is on $C_T(e')$, the supporting cycle of e'. We need to show:

(i) $w(e') \leq w(e)$

(ii) if $w(e') = w(e)$, then e is redundant in $C_T(e')$

Assume the vertices of $C_T(e')$ are $v_1, ..., v_s$, where $e' = (v_1, v_s)$ and $e = (v_m, v_{m+1})$. To simplify notation, we may assume without loss of generality that $e \neq (v_{s-1}, v_s)$ (otherwise, we may reverse the order of the vertices along $C_T(e')$).

(i) We first show that $w(e') \leq w(e)$. Let e_i ($i = 1, ..., m+1$) denote edge (v_i, v_s), and let C_i be its supporting cycle. Let t_i be the unique triangle containing edges e_i and e_{i+1}. By

Lemma 3, e_i is redundant in t_i, for $i = 1, ..., m$. Consider the sequence of triangles $t_1, ..., t_m$. In t_i, $1 \le i \le m-1$, we have $w(e_i) \le w(e_{i+1})$; in triangle t_m, we have $w(e_m) \le w(e)$. Together, we have

$$w(e') = w(e_1) \le w(e_2) \le ... \le w(e_m) \le w(e) \tag{13}$$

(ii) Now assume $w(e') = w(e)$. We can replace the inequalities in (13) by equalities

$$w(e') = w(e_1) = w(e_2) = ... = w(e_m) = w(e) \tag{14}$$

From (14), we conclude that edge e_{i+1} is redundant in triangle t_i, for $i = 1, ..., m-1$; otherwise, we would have $w(e_{i+1}) > w(e_i)$, violating the equality. Similarly, e is redundant in t_m.

Finally, to show that e is redundant in $C_T(e') = C_1$, we prove by induction on j that e is redundant in C_{m-j}, for $j = 0, ..., m-1$.

For $j = 0$, we have to show that e is redundant in C_m. e is redundant in t_m, and e_{m+1} is redundant in its supporting cycle C_{m+1}, thus e is redundant in C_m. Now assume that e is redundant in C_{m-j}. Since e_{m-j} is redundant in t_{m-j-1}, e is also redundant in C_{m-j-1}, which completes the induction. □

Theorem 3 Let G be a minimal network, and assume TD uses a triangle labeling w of G. G is tree decomposable iff TD finds a tree decomposition of G.

Proof: Clear. □

Theorem 5 Given a path-consistent network R, algorithm TL generates a triangle labeling of R in $O(n^3)$ steps.

Proof: The proof is outlined in the text. □

Corollary 3 The tree decomposability of a minimal network G can be decided in $O(n^3)$ steps. Furthermore, if it exists, a tree decomposition of G can be generated in $O(n^3)$.

Proof: Algorithm TD decides whether a tree decomposition exists, and if it does the algorithm generates one (Theorem 3). Since the complexity of generating triangle labeling is $O(n^3)$ and since the complexity of TD without the weight-generation step is also $O(n^3)$, the overall complexity is $O(n^3)$. □

Theorem 6 Algorithm TD is complete for the following networks:

1. Tree-reducible networks
2. Row-convex networks
3. Implicational constraints networks
4. Binary $(0,1)$ networks
5. Distributive networks

Proof: Parts 2, 3, and 4 follow from the fact that row-convex networks [16], implicational constraints networks [8, 1], and distributive networks [12] were shown to be minimal following the application of path-consistency. Also, we already showed that tree-reducible networks that are path consistent are minimal. A tree-reducible network R must have an equivalent tree subnetwork R' containing a subset of its edges. Let us denote by $path(R)$ the network resulting from applying path-consistency to R. Since R is tighter than R', $path(R)$ is tighter than $path(R')$. Since $path(R')$ is minimal and since the two networks are equivalent, $path(R)$ is minimal as well. $\square$