TECHNICAL REPORT
R-146

Tree Decomposition with Applications to Constraint Processing

Itay Meiri, Rina Dechter † and Judea Pearl

Cognitive Systems Laboratory
Computer Science Department
University of California
Los Angeles, CA 90024
itay@cs.ucla.edu, dechter@techsel.bitnet, judea@cs.ucla.edu

Abstract

This paper concerns the task of removing redundant information from a given knowledge base, and restructuring it in the form of a tree, so as to admit efficient problem solving routines. We offer a novel approach which guarantees the removal of all redundancies that hide a tree structure. We develop a polynomial time algorithm that, given an arbitrary constraint network, generates a precise tree representation whenever such a tree can be extracted from the input network; otherwise, the fact that no tree representation exists is acknowledged, and the tree generated may serve as a good approximation to the original network.

1. Introduction

This paper concerns the problem of finding computationally attractive structures for representing constraint-based knowledge.

It has long been recognized that sparse constraint networks, especially those that form trees, are extremely efficient both in storage space and in query processing. A densely-specified network may hide such a desirable structure, and the challenge is to identify and remove redundant links until the natural structure underlying the knowledge base is recovered. The general issue of removing redundancies has been investigated in the literature of relational databases [Maier 1983, Dechter 1987], as well as in the context of constraint networks [Dechter and Dechter 1987]. This paper offers a novel approach which guarantees the removal of all redundancies that hide a tree structure.

Formally, the problem addressed is as follows. Given a constraint network, find whether it can be transformed into a tree-structured network without loss of information; if the answer is positive find such a tree, if the answer is negative, acknowledge failure.

This paper develops a polynomial time algorithm that, given an arbitrary network, generates a tree representation having the following characteristics:

1. The tree represents the network exactly whenever such a tree can be extracted from the input network, and

2. If no tree representation exists, the fact is acknowledged, and the tree generated may serve as a good approximation to the original network.

The algorithm works as follows. 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 algorithm returns a maximum-spanning-tree relative to these weights.

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 obtains. This technique may be regarded as an alternative redundancy-removal scheme to the one proposed in [Dechter and Dechter 1987], accompanied with polynomial complexity and performance guarantees.

2. Preliminaries and nomenclature

We first review the basic concepts of constraint satisfaction [Montanari 1974, Mackworth 1977, Dechter and Pearl 1987].

* This work was supported in part by the National Science Foundation Grant # IRI 8815522 and an Air Force Grant # AFOSR 880177

† Current address: Computer Science Department, Technion - Israel Institute of Technology, Haifa 32000, Israel.

A **network of binary constraints** consists of a set of variables $\{X_1, \ldots, X_n\}$ and a set of binary constraints on the variables. The **domain** of variable X_i, denoted by D_i, defines the set of values X_i may assume. A **binary constraint,** R_{ij}, on variables X_i and X_j, is a subset of the Cartesian product of their domains (i.e., $R_{i,j} \subseteq D_i \times D_j$); it specifies the permitted pairs of values for X_i and X_j.

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

A tuple 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. Two networks with the same variable set are **equivalent** if they represent the same relation.

A binary CSP 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 $i_0 = i, i_1, \ldots, i_m = j$, denoted by $R_{i_0,i_1,\ldots,i_m}$, represents the **composition** of the constraints along the path. 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 $v_1 \in D_{i_1}, \ldots, v_{m-1} \in D_{i_{m-1}}$ such that $R_{i_0,i_1}(x,v_1), R_{i_1,i_2}(v_1,v_2), \ldots,$ and $R_{i_{m-1},i_m}(v_{m-1},y)$.

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 $i_0, i_1, \ldots, i_m$ is consistent, if and only if $R_{i_0,i_m} \subseteq R_{i_0,i_1,\ldots,i_m}$. Similarly, arc (i,j) is consistent if for any value $x \in D_i$, 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 consistent. Any network can be converted into an equivalent arc and path consistent form in time $O(n^3)$(1) [Mackworth and Freuder 1985]. In this paper we assume all networks are arc and path consistent.

Not every relation can be represented by a binary constraint network. The best network that approximates 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.

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

3. Problem statement

We now define the **tree decomposability problem**. First, we introduce the notion of **tree decomposition.**

Definition. 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., 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). R is **tree reducible** if there exists a tree T such that R is decomposable by T, and for all $(i,j) \in T$, $T_{ij} = R_{ij}$, namely the constraints in T are taken *unaltered* from R.

The **tree decomposability problem** for networks is defined as follows. Given a network R, decide if R is tree decomposable. If the answer is positive find a tree decomposition of R, else, acknowledge failure. The **tree reducibility problem** is defined in a similar way. A related problem of decomposing a relation was treated in [Dechter 1987], and will be discussed in Section 6.

Example 1. Consider a relation ρ_1 shown in Figure 1. The minimal network is given by

$$M_{A,B} = M_{A,C} = M_{B,C} = \{00, 11\}$$

$$M_{A,D} = M_{B,D} = M_{C,D} = \{00, 10, 11\},$$

where constraints are encoded as lists of permitted pairs. Any tree containing two edges from $\{AB, AC, BC\}$ is a tree decomposition of M; for example, $T_1 = \{AB, AC, AD\}$ and $T_2 = \{AB, BC, BD\}$. M is also tree reducible, since the link constraints in these trees are identical to the corresponding constraints in M.

A	B	C	D
0	0	0	0
1	1	1	0
1	1	1	1

Figure 1. ρ_1 – a tree-decomposable relation.

Example 2. Consider a relation ρ_2 shown in Figure 2. $T = \{AB, AC, AD, AE\}$ is the only tree decomposition of

ρ_2.

A	B	C	D	E
0	1	1	0	0
0	1	1	0	1
0	1	1	1	0
0	1	1	1	1
1	0	0	1	1
1	0	1	1	1
1	1	0	1	1
1	1	1	1	1

Figure 2. ρ_2 – a tree-decomposable relation.

The rest of the paper is organized as follows. Sections 4 and 5 describe the tree decomposition scheme, while Section 6 presents extensions and ramifications of this scheme. Proofs of theorems can be found in [Meiri, Dechter and Pearl 1990].

4. Tree decomposition schemes

Tree decomposition comprises two subtasks: searching for a skeletal spanning tree, and determining the link constraints on that tree. If the input network is minimal, the second subtask is superfluous because, clearly, the link constraints must be taken unaltered from the corresponding links in the input network, namely, decomposability coincides with reducibility. We shall, therefore, first focus attention on minimal networks, and postpone the treatment of general networks to Section 6. Our problem can now 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 [Even 1979]), 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 $P = i_0, i_1, \ldots, i_m$. Recall that the direct constraint R_{i_0,i_m} is tighter than the path constraint $R_{i_0,i_1,\ldots,i_m}$. If the two constraints are identical we say that edge (i,j) is **redundant** with respect to path P; it is also said to be redundant in the cycle C consisting of nodes $\{i_0, i_1, \ldots, i_m\}$. If the direct constraint is strictly tighter than the path constraint, we say that (i,j) is **nonredundant** with respect to P (or nonredundant in C). Another interpretation of redundancy is that any instantiation of the variables $\{i_0, i_1, \ldots, i_m\}$ 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 which violates R_{i_0,i_m}.

Definition. Let T be a tree, and let $e = (i,j) \notin T$. The unique 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 if and only if 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 1. 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. On the other hand, $T_2 = \{AD, BD, CD\}$ is not a tree decomposition since edge AB is nonredundant in triangle $\{A, B, D\}$ (indeed, the tuple $(A = 1, B = 0, C = 0, D = 0)$ is a solution of T_2, but is not part of ρ_1).

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 leads to 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. This algorithm, called TD-1, is depicted in Figure 3.

Algorithm TD-1

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

Figure 3. TD-1 – A tree decomposition algorithm.

Theorem 2. Let G be a minimal network. Algorithm TD-1 produces a tree T if and only if G is 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 which might prevent the algorithm from reaching a tree structure is that of a **stiff cycle**, i.e., one in which every edge is nonredundant (e.g. cycle $\{B, D, C, E\}$ in Example 2). It can be shown, however, that one of the edges in such a cycle must be redundant in another cycle.

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

Lemma 1. Let G be a path consistent network and let $e = (i_0, i_m)$ be an edge redundant in cycle $C = \{i_0, i_1, \ldots, i_m\}$. If $C' = \{i_0, i_1, \ldots, i_k, i_{k+l}, \ldots, i_m\}$ is an interior cycle created by chord (i_k, i_l), then e is redundant in C'.

Lemma 2. 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 3. 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 $T' = T - \{e\} \cup \{e'\}$.

Algorithm TD-1, though 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 which overcomes this difficulty.

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 TD-1, whereby the criterion for removing an edge is essentially precomputed. To guide TD-1 in selecting redundant edges, we first impose an ordering on the edges, in such a way that nonredundant edges will always attain higher ranking than redundant ones. Given such ordering, we do not remove edges of low ranking, but apply the dual method instead, and construct a tree containing the preferred edges by finding a maximum weight spanning tree (MWST) relative to the given ordering. This idea is embodied in the following scheme.

Definition 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 if and only if T is a MWST of G with respect to w.

Finding a tree labeling essentially solves the tree decomposability problem, simply following the steps of algorithm TD-2 shown in Figure 4. TD-2 stands for a family of algorithms, each driven by a different labeling. Steps 2-4 can be implemented in $O(n^3)$: Step 2 can use any MWST algorithm, such as the one by Prim, which is $O(n^2)$ (see [Even 1979]); Steps 3-4, deciding whether G is decomposable by T, are $O(n^3)$ as explained in Section 4.

Algorithm TD-2

1. $w \leftarrow$ tree labeling of G;
2. $T \leftarrow$ 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;

Figure 4. TD-2 – A polynomial tree decomposition algorithm.

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 satisfying this condition. As a result, the total running time of TD-2 is bounded by $O(n^3)$.

Definition. 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

$$(i)\ w(e') \le w(e). \qquad (1)$$

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

Theorem 3. Let w be any labeling of a minimal network G. w is a tree labeling if and only if w is a redundancy labeling.

The merit of Theorem 3 is that it is often easier to test for redundancy labeling than for the ultimate objective of tree labeling. Having established this equivalence, the next step is to construct a labeling that satisfies conditions

(1) and (2).

Definition. 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

$$w(e_1) \leq w(e_2)\,,\; w(e_1) \leq w(e_3). \qquad (3)$$

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

$$w(e_1) < w(e_2). \qquad (4)$$

Conditions (3) and (4) will be called **triangle constraints.**

Illustration. Consider the minimal network of Example 2. Analyzing redundancies relative to all triangles leads to the triangle constraints depicted in Figure 5. Each node in the figure represents an edge of the minimal network, and an 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 easily constructed 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 a MWST relative to these weights, a property that we will show to hold in general.

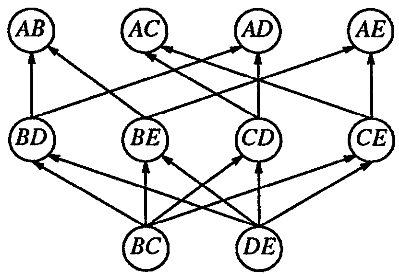

Figure 5. Triangle constraints for Example 2.

Clearly, conditions (3) and (4) are easier to verify as they involve only test on triangles. In Theorem 5 we will indeed show that they are sufficient to constitute a redundancy labeling, hence a tree labeling. Moreover, a labeling satisfying (3) and (4) is easy to create primarily because, by Theorem 4, such a labeling is guaranteed to exist for any path consistent (hence 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 $w(a) > w(b)$.

Theorem 4. Any path consistent network admits a triangle labeling.

The idea behind triangle labelings is that all redundancy information necessary for tree decomposition can be extracted from individual triangles rather than cycles. By Lemma 1, 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 which are nonredundant relative to others. The correctness of the next theorem rests on these considerations.

Theorem 5. Let G be a minimal network, and let w be a labeling of G. If w is a triangle labeling then it is also a redundancy labeling.

By Theorems 3 and 5, if the network is minimal any triangle labeling is also a tree 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 TLA, shown in Figure 6, accomplishes this task.

Algorithm TLA

1. create an empty directed graph $G_1 = (V_1,E_1)$ with $V_1 = E$;
2. for each triangle $t = \{e_i, e_j, e_k\}$ in G do
3.   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;
4. $G_2 = (V_2,E_2) \leftarrow$ superstructure of G_1;
5. compute a topological ordering w for V_2;
6. for $i := 1$ to $|V_2|$ do
7.   for each edge e in C_i do
8. $w(e) \leftarrow w(C_i)$;

Figure 6. TLA – an algorithm for constructing a triangle labeling.

Let us consider the TLA algorithm 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 $w(u) \le w(v)$ 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.

Consider a pair of nodes, u and v, in G_1. It can be verified that if they belong to the same strongly-connected component (i.e., they lie on a common directed cycle), 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 [Even 1979]. 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 [Even 1979]).

It is well-known that the superstructure forms a DAG (directed acyclic graph), moreover, 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 5) 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 (Steps 6-8), 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 $O(n^3)$.

Illustration. Consider Example 1. There are two strongly-connected components in G_1:

$$C_1 = \{AD, BD, CD\}$$

and

$$C_2 = \{AB, AC, BC\}.$$

There are edges going only from C_1 to C_2. Thus, assigning weight 1 to all edges in C_1 and weight 2 to all edges in C_2 constitutes a triangle labeling. Consider Example 2, for which G_1 is shown in Figure 5. Note that $G_2 = G_1$, that is, every strongly-connected component consists of a single node. Assigning weights in the ranges 1-2, 3-6 and 7-10 to the bottom, middle and top layers, respectively, constitutes a triangle labeling.

6. Extensions and Ramifications

6.1. Decomposing a relation

Given a relation ρ, we wish to determine whether ρ is tree decomposable. We first describe how TD-2 can be employed to solve this problem, and then compare it with the solution presented in [Dechter 1987].

We start by generating the minimal network M from ρ. We then apply TD-2 to solve the decomposability problem for M. If M is not tree decomposable, ρ cannot be tree decomposable; because otherwise, there would be a tree T satisfying $\rho = rel(T) \subset rel(M)$, violating the minimality of M [Montanari 1974]. 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 if and only if $|\rho| = |rel(T)|$. Generating M takes $O(n^2 |\rho|)$ operations, while $|T|$ can be computed in $O(n)$ time [Dechter and Pearl 1987]; thus, the total time of this method is $O(n^2 |\rho|)$.

An alternative solution to the problem was presented in [Dechter 1987]. It computes for each edges a numerical measure, w, based on the frequency that 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

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

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

Comparing the two schemes, our method 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.

6.2. Reducing a network

Given an arc and path consistent network R, we wish to determine whether R is tree reducible. This problem admits TD-2 directly, since it can be shown that any path consistent network is tree reducible only when it is minimal. Thus, if TD-2 returns failure, we are assured that R is not tree reducible (though it could still be tree decomposable).

6.3. Removing redundancies from a network

Given a network R (not necessarily tree decomposable), we wish to to remove as many redundant edges as possible from the network. Our scheme provides an effective heuristics, alternative to that of [Dechter and Dechter 1987]. We first apply the TD-2 algorithm and, in case the tree generated does not represent the network precisely, we add nonredundant edges until a precise representation obtains.

6.4. Approximating a Network

Given a network R, find a tree network which constitutes a good approximation of R. The tree T generated by TD-2 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: The tree T generated by TD-2 is **most specific** in the following sense: no other tree T', extracted form the network, satisfies $rel(T') \subset rel(T)$.

Although we could find no proof yet, the conjecture has managed to endure all attempts to construct a counterexample.

7. Conclusions

We have addressed the problem of decomposing a constraint network into a tree. We have developed a tractable decomposition scheme which requires $O(n^3)$ time, and solves the problem for minimal networks. The technique maintains its soundness when applied to an arbitrary network, and is guaranteed to find a tree decomposition if it can be extracted from the input network without altering the link constraints. The main application of our scheme lies in preprocessing knowledge bases and transforming them into a very effective format for query processing. Other applications are in guiding backtrack search by tree relaxation of subproblems. Finally, we envision this technique to be useful in inductive learning; especially, for learning and generalizing concepts where instances are observed sequentially. The tree generated by TD-2 provides one of the simplest descriptions consistent with the observed data, and at the same time it is amenable to answer queries of subsumption and extension.

References

Dechter A. and Dechter R. 1987. Removing Redundancies in Constraint Networks. In Proceeding of AAAI-87, Seattle, WA.

Dechter R. 1987. Decomposing a Relation into a Tree of Binary Relations. In Proceedings of 6th Conference on Principles of Database Systems, San Diego, CA, 185-189. To appear in *Journal of Computer and System Science*, Special Issue on the Theory of Relational Databases.

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

Even S. 1979. *Graph Algorithms*. Computer Science Press, Rockville, Md.

Freuder E. C. 1982. A Sufficient Condition of Backtrack-Free Search, *JACM* 29(1), 24-32.

Mackworth A. K. 1977. Consistency in Networks of Relations. *Artificial Intelligence* 8(1), 99-118.

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

Maier D. 1983. *The Theory of Relational Databases*. Computer Science Press, Rockville, Md.

Meiri I., Dechter R. and Pearl J. 1990. Tree Decompositions with Applications to Constraint Processing. Technical Report R-146, Cognitive Systems Lab., University of California, Los Angeles.

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