Coupling CSP Decomposition Methods and Diagnosis Algorithms for Tree-Structured Systems*

Markus Stumptner
University of South Australia,
Advanced Computing Research Centre
5095 Mawson Lakes SA, Adelaide, Australia
mst@cs.unisa.edu.au

Franz Wotawa [f]
Technische Universitat Graz
Institute for Software Technology (1ST)
8010 Graz, Inffeldgasse 16b/2, Austria
wotawa@ist.tugraz.at

Abstract

Decomposition methods are used to convert general constraint satisfaction problems into an equivalent tree-structured problem that can be solved more effectively. Recently, diagnosis algorithms for tree-structured systems have been introduced, but the prerequisites of coupling these algorithms to the outcome of decomposition methods have not been analyzed in detail, thus limiting their diagnostic applicability. In this paper we generalize the TREE* algorithm and show how to use hypertree decomposition outcomes as input to the algorithm to compute the diagnoses of a general diagnosis problem.

1 Introduction

The development of effective algorithms for diagnosing large and complex systems remains one of the key issues in model-based reasoning. Nonetheless, apart from various additional optimizations and control strategies (e.g., [de Kleer, 1991]) the main architectures for consistency-based diagnosis systems, TMS [de Kleer and Williams, 1987] and hitting sets computation [Reiter, 1987], have remained remarkably stable since the late 1980s. Over the last years though, a number of algorithms were published that exploited advantageous structural properties of the systems to be diagnosed for significant computational speedups [Fattah and Dechter, 1995; Stumptner and Wotawa, 2001], enabling the very fast diagnosis of large tree-structured systems.

It was recognized early on that the combination of tree-oriented algorithms could in principle be combined with a second class of algorithms aiming at the decomposition of cyclic problems to equivalent tree-structured problems so that the faster solution algorithms could be brought to bear [Fattah and Dechter, 1995;Darwiche, 1998]. Again, this direction profits from recent new results relating the different decomposition algorithms [Gottlob *et al.,* 2000] and analyzing their performance [Gottlob *et al. ,* 2002]. This paper joins the two strands by tying the link between the recent work on problem decomposition and the TREE* algorithm [Stumptner and Wotawa, 2001].

This work was partially supported by the Austrian Science Fund (FWF) under project grants P15163-1NF and P 15265-INF.

Authors are listed in alphabetical order.

This combination is aided by the fact that both subareas tend to use Constraint Satisfaction Problems (CSPs) as their representation of choice. The diagnosis computation work [Fattah and Dechter, 1995; Stumptner and Wotawa, 2001; Mauss and Tatar, 2002] focuses on the relational combination of the different constraint relations, while the problem reformulation work views CSPs as (often cyclic) hypergraphs that are broken down into tree structures.

A CSP (V, *D, C)* comprises a set of variables *V,* their domains D, and a set of constraints *C.* A constraint C_i is tuple (S_i, T_i) where $S_i \subseteq V$ is its scope and T_i is a set of tuples of values for the variables in S_i. We assume two functions *tpl* and *scope* on constraints that return the set of tuples and the scope respectively. Each constraint C_i restricts the possible values of the variables with respect to values of other variables in the same scope.For example, the digital circuit from Figure 1 can be represented by the following constraints':

N_1

a	f	g
1	1	1
0	0	0

N_2

f	b	h
0	0	0
0	1	1
1	0	1
1	1	1

N_3

g	c	i
0	0	0
0	1	0
1	0	0
1	1	1

N_4

h	i	j
0	0	0
0	1	1
1	0	1
1	1	1

A solution of a CSP is an assignment of values to all variables that satisfy the given constraints. All values must be elements of their variable's domain. The variable assignment $b = 1, a = 0, c = 0, f = 0, g = 0, h = 1, i = 0, j = 1$ is a solution for the above CSP whereas $b = 1, a = 0, c = 0, f = 0, g = 0, h = 1, i = 0, j = 0$ is not because constraint N_4 is not satisfied.

A standard approach for computing solutions for CSPs are backtracking algorithms, which are in the worst case exponential in the number of variables. However, for a specific

'Note that not all constraints arc components, N_1 is the tabular representation of an equality constraint between connections).

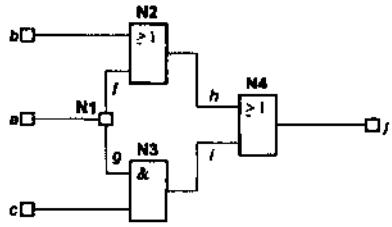

Figure 1: A famous small combinational circuit

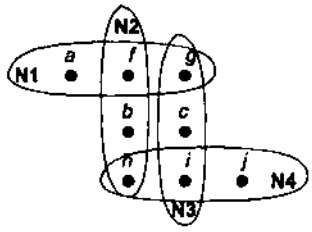

Figure 2: A hypergraph representing the constraints

class of CSPs a solution can be computed in polynomial time. This class comprises all CSPs that have an acyclic corresponding hypergraph[2]. A hypergraph of a CSP can be easily constructed by mapping all variables of the CSP to vertices and the constraint scopes to hyperedges. A CSP with an acyclic hypergraph can be solved effectively in a backtrack-free manner by first traversing the graph from the leafs to the root and computing possible value tuples and secondly, traversing the graph from the root to the leafs and selecting one tuple of a node as a solution. The hypergraph corresponding to Fig. 1 is cyclic (see Figure 2).

In the rest of the paper, we recapitulate decomposition methods, present a version of TREE* that fits these methods, show the interaction of decomposition and TREE*, and present an extension to the algorithm that can be used with extended domains as presented in [Mauss and Tatar, 2002].

2 Decomposition methods

Several different decomposition methods have been published, most recently the hypertree decomposition method [Gottlob *et al.,* 1999a]. This and other structure-based decomposition methods make compute a tree-structured systems from general CSPs by elimination of cycles from the CSP. A cycle can be eliminated by applying the relational algebra join operation to the constraints on the cycle. However, blindly joining all constraints in the cycle can result in exponential costs for computing the tuples of the joined constraint, and a key property of the different decomposition methods is the different techniques they use to select the constraints to be joined. Gottlob et al. [2000] compared different decomposition methods with respect to their width, i.e., the maximum number of constraints to be joined, and found that hypertree decomposition is superior with respect to the width. Since hypertree decomposition can be seen as a generalization of the other techniques, we restrict our examination here to hypertree decomposition. Note that the result of a decomposition method is always a hypertree.

In [Gottlob *et al.,* 2000] hypertree decomposition is characterized as follows. A hypertree of a hypergraph if is a triple (T, χ, λ), where $T = (N, E)$ is a rooted tree with vertices *N* and edges E, χ and λ are labeling functions which associate to each vertex $p \in N$ a set of variables $\lambda(p) \subseteq var(H)$ and a set of constraints $\chi(p) \subseteq edges(H)$. We further define χ for a subtreb $T' = (N', E')$ of T: $\chi(T') = \bigcup_{v \in N'} \chi(v)$, d for any $p \in N$, T_p denotes the subtree of *T* rooted at p. We denote the root of a hypertree by *root(T)* .

Based on the above definitions, the hypertree decomposition of a hypergraph *H* is defined as a hypertree *HD* — (T, χ, λ) where *T* — (TV, *E)* which satisfies the following conditions:

1. For each edge $h \in$ *edges(H),* there exists a $p \in N$ such that *var(h)* $\subseteq \chi(p)$
2. For each variable $Y \in var(H)$, the set $\{p \in N | Y \in \backslash(p)\}$ induces a connected subtree of *T.*
3. For each $p \in N, \chi(p) \subseteq var(\lambda(p))$.
4. For each $p \in N, var(\lambda(p)) \cap \chi(T_p) \subseteq \chi(p)$

Moreover, we say that *h* ∈ *edges(H)* is strongly covered in H*D* if there exists a vertex *p* ∈ *N* such that *var(h)* $\subseteq \chi$ (p) and $h \in \lambda(p)$. A hypertree decomposition *HD* of a hypergraph H is a complete decomposition of H if every edge of ff is strongly covered in H*D.* Note that it is always possible to make an incomplete decomposition complete by adding new vertices to the decomposition. Gottlob et al. [1999b] gave an algorithm for computing (complete) hypertree decompositions.

Note that unlike other decomposition methods such as biconnected component decomposition and hinge decomposition, there is in general no unique hypertree decomposition of a given CSP. Figure 3 shows four possible hypertree decompositions for the hypergraph depicted in Figure 2.

3 Diagnosis with TREE*

To be self contained we briefly recapitulate the TREE* algorithm. Stumptner and Wotawa [2001] introduced the TREE* algorithm as an extension of the TREE algorithm. Both algorithms work on tree-structured constraint systems. As opposed to TREE which requires the constraints to be mathematical functions, TREE* imposes no limitations on the constraints. TREE* uses the following auxiliary functions associated to constraints: *constr* denotes all tuples of the constraint, *val* denotes the tuples remaining after the application of TREE*, *diags* denotes the diagnoses that correspond to a given tuple. Accordingly, a CSP that is to be used for diagnosis purpose has to represent not only the tuples for each constraint but also the diagnoses that correspond to each tuple. For example, the small circuit from Figure 1 can be represented by the following CSP:

[2] A hypergraph (N, E) consists of a set of vertices N and a set of hyperedges E such that $e \subseteq N$ for each $e \in E$

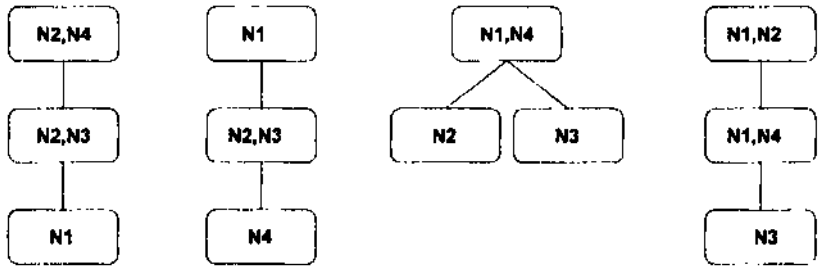

Figure 3: Alternate hypertree decompositions of Figure 2

N_1

a	f	g	*diags*
1	1	1	{{}}
0	0	0	{{}}

N_2

f	b	h	*diags*
0	0	0	{{}}
0	1	1	{{}}
1	0	1	{{}}
1	1	1	{{}}
x	x	x	$\{\{N_2\}\}$

N_3

g	c	i	*diags*
0	0	0	{{}}
0	1	0	{{}}
1	0	0	{{}}
1	1	1	{{}}
x	x	x	$\{\{N_3\}\}$

N_4

h	i	j	*diags*
0	0	0	{{}}
0	1	1	{{}}
1	0	1	{{}}
1	1	1	{{}}
x	x	x	$\{\{N_4\}\}$

The tuples of form <x x... x> in the above tables are intended to match all tuples that are not explicitly given in the table. Note that the constraint $N\backslash$ represents a connection and is therefore assumed to be always correct. Hence, it always returns the empty set as a diagnosis.

The original description of the TREE* algorithm was based on the underlying assumption that the leaf vertices of the tree correspond only to one diagnosis component. This assumption has an impact on the description of the algorithm, but not on the empirical results. If the constraints given to the algorithm do not satisfy this requirement, diagnoses of a size (i.e., number of faulty components) larger than specified might be returned because there is no way to remove these diagnoses from the set of tuples. The requirement did not constitute a hindrance since [Stumptner and Wotawa, 2001] did not deal with CSPs resulting from decomposition. To explicitly generalize TREE* to interact with decomposition algorithms, we here present a modified version that removes this requirement.

The following operations are used by TREE*: semi-join ($\ltimes$) for constraints, Cartesian product (x) for combining two sets of diagnoses, i.e., $DS_1 \times DS_2 = \{d_1 \cup d_2 | d_1 \in DS_1 \wedge d_2 \in DS_2\}$, and cardinality restriction (|) for removing diagnoses from a set of diagnoses with a size greater than the given value, i.e., $DS|s = \{d | d \in DS \wedge |d| <= s\}$.

Algorithm TREE* *(HD,* p, *diagSize)*
Computes all diagnoses up to a pre-specified size for a given tree-structured diagnosis system.
Input: A decomposition *HD* with edges *E,* the root vertex p, and the pre-specified diagnosis size *diagSize.*
Output: The diagnoses for each value tuple.

(1) $V \leftarrow constr(p) \ltimes observed(p)$
(2) **for** $s \in V$ **do**
(3) $\quad diags(s) \leftarrow diags(s)|diagSize$
(4) $\quad$ **if** $diags(s) = \emptyset$ **then**
(5) $\quad\quad V \leftarrow V \setminus \{s\}$
(6) $\quad$ **end if**
(7) **end for**
(8) **for** $(p, q) \in E$ **do**
(9) $\quad$ TREE*(q)
(10) $\quad V \leftarrow V \ltimes val(q)$
(11) $\quad$ **for** $s \in V, t \in val(q)$ **such that** s, t **join do**
(12) $\quad\quad diags(s) \leftarrow (diags(s) \times diags(t))|diagSize$
(13) $\quad\quad$ **if** $diags(s) = \emptyset$ **then**
(14) $\quad\quad\quad V \leftarrow V \setminus \{s\}$
(15) $\quad\quad$ **end if**
(16) $\quad$ **end for**
(17) **end for**
(18) $val(p) \leftarrow V$

The TREE* algorithm is called using the root p of the hypertree, i.e., the result of a decomposition, as argument. After execution, the computed diagnoses can be found in the *diay* column of the tuples associated with the root *p.*

The TREE* algorithm correctly computes all diagnoses up to the given size if the following requirements arc fulfilled:

1. Every constraint is at least used in one of tree vertices.
2. The induced subtree for every variable is connected.

Theorem 1 (Correctness) *The TREE* algorithm correctly computes diagnoses up to the pre-specified size.*

Proof. The proof is by induction over the size of the tree.
Base step For each leaf of the tree only lines (I)-(7) and (18) of TREE* are executed. In these lines, all tuples that contradict the given observations and all diagnoses larger than the specified size are removed. Tuples with no corresponding diagnoses are removed. These steps do not prevent TREE* from computing a diagnosis. Hence, all diagnoses (up to the pre-specified size) are computed for leafs.

Hypothesis We assume that TREE* correctly computes diagnoses for trees of size smaller than *n.*

Induction step We now prove that TREE* correctly computes the diagnoses for trees of size *n.* The steps (I)-(7) only reduce the number of diagnoses. Tuples that contradict the observations are removed. Diagnoses that are larger than the pre-specified values are removed and tuples with a corresponding empty diagnosis set are also removed because they have no influence on the result. In step (9) the TREE* algorithm is called recursively. Because of our induction hypothesis, the algorithm returns correct diagnoses for each tuple. It remains to prove that the combination of these tuples with the tuple of the current node is done correctly. Since every tuple that joins (line (11)) is considered and since that every diagnosis is combined with every diagnosis of the child vertices (line (12)), this process must lead to a correct result. Hence, TREE* correctly computes the diagnoses in step *n* as well. □

Using the same arguments as above we can show that TREE* allows for computing all diagnoses providing that all the diagnoses for each vertex of the hypertree are computed. We omit the actual proof here.

Theorem 2 (Completeness) *The TREE* algorithm computes all diagnoses up to the pre-specified size.*

The remaining issue now is to determine the conditions that a decomposition method must satisfy in order to be used with TREE*. Moreover, we examine the computation of the join relation for those tree vertices that comprise more than one constraint.

4 Decomposition and Joining of Constraints

As an example we take the hypertree decomposition result from Figure 3(b). In order to apply TREE*, we first have to compute the tuples for the constraints that occur in one vertex, e.g., *{N2, N3}.* The tuples can be computed by first joining the constraints, and second computing the diagnoses *diags* for each tuple of the joined constraint by combining the diagnoses associated with the corresponding tuples of the original constraints. The following algorithm computes the join relation for the constraint of a hypertree vertex *p.*

Algorithm JoinRelation *(p)*
Computes the join for a hypertree vertex
Input: A hypertree vertex *p.*
Output: The *coiistr* and *diags* function of the vertex *p.*

(1) $constr(p) \leftarrow c_1 \bowtie \ldots \bowtie c_k$ **where** $\{c_1, \ldots, c_k\} = \lambda(p)$.
(2) **for** $t \in constr(p)$ **do**
(3) $\quad diags_p(t) \leftarrow \{\{\}\}$
(4) $\quad$ **for** $c \in \lambda(p)$ **do**
(5) $\quad\quad$ **if** $scope(c) \subseteq \chi(p)$ **then**
(6) $\quad\quad\quad diags_p(t) \leftarrow diags_p(t) \times diags_c(\pi_{scope(c)} t)$
(7) $\quad\quad$ **end if**
(8) $\quad$ **end for**
(9) **end for**

In the above algorithm the *diags* function is indexed with the corresponding vertex of the hypertree or the corresponding constraint. The function ⋈ stands for relational join and for relational projection. Diagnoses are only combined if a constraint is fully captured by the given vertex of the hypertree. Otherwise, it is not considered. Since every constraint must be fully captured by at least one hypertree vertex, no information is lost by this procedure.

For example, the join of constraints N_2 and N_3 would lead to the following constraint $[N_2, N_3]$ with 25 tuples (when retaining the V constants, and of course 64 otherwise):

f	b	h	g	c	i	*diags*
0	0	0	0	0	0	{{}}
0	0	0	0	1	0	{{}}
0	0	0	1	0	0	{{}}
0	0	0	1	1	1	{{}}
0	0	0	x	x	x	{{N_3}}
0	1	1	0	0	0	{{}}
0	1	1	0	1	0	{{}}
0	1	1	1	0	0	{{}}
0	1	1	1	1	1	{{}}
0	1	1	x	x	x	{{N_3}}
1	0	1	0	0	0	{{}}
1	0	1	0	1	0	{{}}
1	0	1	1	0	0	{{}}
1	0	1	1	1	1	{{}}
1	0	1	x	x	x	{{N_3}}
1	1	1	0	0	0	{{}}
1	1	1	0	1	0	{{}}
1	1	1	1	0	0	{{}}
1	1	1	1	1	1	{{}}
1	1	1	x	x	x	{{N_3}}
x	x	x	0	0	0	{{N_2}}
x	x	x	0	1	0	{{N_2}}
x	x	x	1	0	0	{{N_2}}
x	x	x	1	1	1	{{N_2}}
x	x	x	x	x	x	{{N_2, N_3}}

If, as in this example, the join operation is a Cartesian Product, the resulting relation is of course very large. However, during the computation of diagnoses using TREE*, many tuples can generally be eliminated because of the given observations and the pre-specificd maximum diagnosis size.

To illustrate this, assume now that we have a set of observations $OBS = \{a = 1, b = 0, c = 1, j = 0\}$ and that we arc searching only for single diagnoses, i.e., *diagSize* — 1. Using the decomposition from Figure 3(b) TREE* is first called with the vertex (constraint) *[N1].* After executing lines (1)-(7) (including the semijoin that results in the removal of tuples that do not fit the operations) the constraint of [N_1] is given by:

a	f	g	*diags*
1	1	1	{{}}

TREE* is then recursively called on vertex $[N_2, N_3]$, resulting in the following relation, which is substantially smaller than the computed join relation for this vertex, even after ail *x* entries of the original tables have been replaced by either 0 or 1, while avoiding duplicated entries (first table), then we again recursively call TREE* which leads to the computation of the following relation for vertex $[N_4]$ (second table):

f	b	h	g	c	i	*diags*
0	0	0	0	1	0	{{}}
0	0	0	1	1	1	{{}}
0	0	0	0	1	1	{{N_3}}
0	0	0	1	1	0	{{N_3}}
1	0	1	0	1	0	{{}}
1	0	1	1	1	1	{{}}
1	0	1	0	1	1	{{N_3}}
1	0	1	1	1	0	{{N_3}}
0	0	1	0	1	0	{{N_2}}
1	0	0	0	1	0	{{N_2}}
0	0	1	1	1	1	{{N_2}}
1	0	0	1	1	1	{{N_2}}

h	i	j	*diags*
0	0	0	{{}}
0	1	0	{{N_4}}
1	0	0	{{N_4}}
1	1	0	{{N_4}}

In the next step TREE* continues the computation at line (10) for vertex *[N_2, N_3]*. After combining the diagnoses and removing the tuples with diagnoses larger than *diagSize* we get the following relation for [N_2, N_3]:

f	b	h	g	c	i	*diags*
0	0	0	0	1	0	{{}}
0	0	0	1	1	1	{{N_4}}
0	0	0	1	1	0	{{N_3}}
1	0	1	0	1	0	{{N_4}}
1	0	1	1	1	1	{{N_4}}
1	0	0	0	1	0	{{N_2}}

And finally for the vertex [N_1]:

a	f	g	*diags*
1	1	1	{{N_4}}

Hence, only the diagnosis *{N4}* is a single diagnosis for our example.

5 Putting it all together

In order to make use of TREE* for general CSPs, we first apply a decomposition method and then apply the algorithm to the resulting acyclic problem. In this section we summarize the requirements placed on the decomposition method.

Theorem 3 *TREE* computes all consistent diagnoses for a given (possibly cyclic) CSP (V,D, C), if it is applied to the decomposition HDof(V,D,C) that was produced by a decomposition method h with the following properties:*

1. *The decomposition result, i.e., a hypertree HD, must be complete.*
2. *The vertices that use a given variable $Y \in V$ must form a connected subtree of the hypertree HD.*

Proof (sketch): We consider each condition in turn. As discussed in the previous section, the decomposition method must produce a complete decomposition, i.e., every constraint must be strongly covered, as this is a prerequisite for the correct working of the join algorithm. Concerning condition 2, assume that this condition were not satisfied, i.e., that for some *Y*, $\{p \in N | Y \in \chi(p)\}$ does not induce a connected subtree. This implies either that (if we have no sub-tree of the resulting hypertree) the decomposition is not cyclic, or that there exists a variable that is used in both sub-trees but not in the parent. In the latter case, the process that computes a solution cannot view the sub-trees as independent problems any more, and TREE* will fail to compute a correct outcome. The other conditions of the hypertree decomposition can be relaxed without affecting the result. If condition 3 is not obeyed, that means some nodes contain variables that are not constrained. This may affect the efficiency of the algorithm (because the node relations include a cartesian product with the values of those variables) but not the correctness. If condition 4 is not obeyed, this means that some variable *z* such that $z \in \chi(p)$ **for a vertex** $p \in N$ is not contained in the constraint associated with p, i.e. in relational terms *z* has been projected away. This means that some subtree of *T* is going to be less restrictive in execution, leading to excess tuples. However, since the decomposition is required to be complete, all constraints that contain *z* must exist in unprojected form somewhere in *T*. □

These two conditions are true for hypertree decomposition [Gottlob *et ai,* 1999a], biconnected components [Freuder, 1985], hinge decomposition [Gyssens *et al.,* 1994], and tree clustering [Dechter and Pearl, 1989].

The following algorithm summarizes the combined use of TREE* and a decomposition method to compute the diagnoses for any CSP *C* of the form described in Section 3. Given a CSP C, using any decomposition method that fulfills the properties of Theorem 3.

(1) apply decomposition to *C,* i.e., compute a hypertree (TV, *E)* for the corresponding hypergraph of *C*
(2) for $v \in N$ do JoinRelation(f) end for;
(3) *TREE*(root(T));*
(4) return diags(roof(T'));

The pre-compilation performance of the overall algorithm is that of the decomposition algorithm (examined in [Gottlob *et al.,* 2002]) together with the required costs for joining constraints using the JoinRelation algorithm. The diagnosis time is the running time of TREE*. Performance and scalability TREE* have been studied in [Stumptner and Wotawa, 2001]. Note that the TREE* run time depends on the size of the relations that are stored in the vertices of the resulting hypertree. This size depends on the number of constraints that must be joined, and this in turn corresponds to the width of the decomposed system. Hence, using a decomposition method that provides a smaller width leads to hypertrees where TREE* performs better.

6 Extension to Intensional Relations

In [Stumptner and Wotawa, 2001], we mentioned the possibility to use other than extensional relations for constraint specifications for the TREE and TREE* algorithm, e.g., computed functions or equations with infinite domains. Such domains require a different interpretation of the operators that are used for joining the relations associated with nodes in the tree. Conceptually, however, nothing is changed since, as we will show, the definition of the TREE* algorithm fits the requirements. We show this by adopting the notation used for the basic computational operations of the *aggregation paradigm* described in [Mauss and Tatar, 2002].

The Rich Constraint Languages approach described in [Mauss and Tatar, 2002] consists of three inference procedures, which are applied to a set of constraints *It.* The procedure isConsistent produces a proof tree (called *aggregation tree)* that derives consistency or inconsistency. If consistent, then, applied to the root of this tree, the procedure solve computes (nondetcrministically) a solution that assigns a value to every variable in R. If not, the procedure XC1 (and its extension XE1) computes the set of minimal conflicts for *A.*

The aggregation operation that gives the tree its name consists of joining two constraints A and B (expressed as relations or equations) and then projecting out all variables except the set of variables X needed for joining to other constraints: $C \leftarrow \pi(A \wedge B, X)$. If $X = var(A)$, this can be written using the semijoin operator that we have used above: $C \leftarrow A \ltimes B$. Thus, the TREE* algorithm can be changed to approximate the aggregation paradigm purely by letting s and / refer to tuples (for extensional constraints) or equations, and changing line (1) to $V \leftarrow \pi(constr(p) \wedge observed(p), var(constr(p)))$;, and line (10) to $V \leftarrow \pi(V \wedge val(q), var(val(q)))$

The JoinRelation Algorithm is changed by replacing the first line by $constr(p) \leftarrow (\wedge c_i, var(p))$.

In terms of algorithmic structure, the approach of [Mauss and Tatar, 2002] bears many resemblances to the decompose-and-diagnose paradigm used in LFattah and Dechter, 1995; Darwiche, 1998; Stumptner and Wotawa, 2001]. Computationally, there are however significant differences. The outcome of isConsistent is a proof tree whose leaves are the base constraints of the original CSP, not a hypertree whose nodes arc the constraints of a backtrack-free CSP equivalent to the original CSP. The search for a diagnosis consists of running isConsistent, computing minimal conflicts, and the hitting sets. In our case, we compute the decomposition hypertree, then apply the JoinRelation algorithm and finally apply TREE* to compute the diagnoses directly.

While the nondeterministic selection of arbitrary constraints from the given CSP to produce a proof tree is quite effective in general, as indicated by the authors, there are cases where the "width" of the generated problem leads to a drastic growth in intermediate relations. It is this situation where an approach based on hypertree decomposition (which is generally the method with lowest widths) fits best to the strengths of the TREE* algorithm.

7 Conclusion

In this paper we introduced a framework that allows for combining various structure decomposition methods and algorithms for solving tree-structured diagnosis problems. The framework comprises two parts. In the first part, we show how to construct a tree-shaped constraint system that can be used directly by TREE* for computing diagnoses. For this purpose we introduced the join relation of constraints. In the second part, we state the properties of a decomposition method so that it can be combined with TREE*. Finally, we show the suitability of the TREE* framework for the extension to infinite domains and intensional constraints (e.g., equations), by adopting a basic operation from another, not hypertree based framework.

References

[Darwiche, 1998] Adnan Darwiche. Compiling Devices: A Structure-Based Approach. In *Proceedings of the International Conference on Principles of Knowledge Representation and Reasoning,* pages 156-166, 1998.

[de Kleer and Williams, 1987] Johan de Kleer and Brian C. Williams. Diagnosing multiple faults. *Artificial Intelli*g^cr,32(l):97-130,1987.

[de Kleer, 1991] Johan de Kleer. Focusing on probable diagnoses. In *Proceedings of the National Conference on Artificial Intelligence (AAAI),* pages 842-848, Anaheim, July 1991.

[Dechter and Pearl, 1989] Rina Dechter and Judea Pearl. Tree clustering for constraint networks. *Artificial Intelligence,* 38:353-366, 1989.

[Fattah and Dechter, 1995] Yousri El Fattah and Rina Dechter. Diagnosing tree-decomposable circuits. In *Proceedings 14th International Joint Conf. on Artificial Intelligence,* pages 1742 - 1748, 1995.

LFreuder, 1985] Eugene C. Freudcr. A Sufficient Condition for Backtrack-Bounded Search. *Artificial Intelligence,* 32(4):755-761,1985.

[Gottlob et al., 1999a] Georg Gottlob, Nicola Leone, and Francesco Scarcello. Hypertree Decomposition and Tractable Queries. In *Proc. 18th ACM SIGACT SIGMOD SIGART Symposium on Principles of Database Systems (PODS-99), pages* 21-32, Philadelphia, PA, 1999. '

[Gottlob *etal,* 1999b] Georg Gottlob, Nicola Leone, and Francesco Scarcello. On Tractable Queries and Constraints. In *Proc. 12th International Conference on Database and Expert Systems Applications DEXA 2001,* Florence, Italy, 1999.

[Gottlob *et al.,* 2000] Georg Gottlob, Nicola Leone, and Francesco Scarcello. A comparison of structural CSP decomposition methods. *Artificial Intelligence,* 124(2):243-282, December 2000.

|Gottlobe.,2002] Georg Gottlob, Martin Hutle, and Franz Wotawa. Combining hypertree, bicomp, and hinge decomposition. In F. van Harmelen, editor, *Proceedings of the 15 th European Conference on Artificial Intelligence (ECAI-02),* Lyon, July 2002. IOS Press.

[Gyssens *et al,* 1994] Marc Gyssens, Peter G. Jeavons, and David A. Cohen. Decomposing constraint satisfaction problems using database techniques. *Artificial Intelligence,* 66:57-89,1994.

[Mauss and Tatar, 2002] Jakob Mauss and Mugur Tatar. Computing minimal conflicts for rich constraint languages. In F. van Harmelen, editor, *Proc. ECAI,* Amsterdam, 2002. IOS Press.

[Reiter, 1987] Raymond Reiter. A theory of diagnosis from first principles. *Artificial Intelligence,* 32(l):57-95,1987.

[Stumptner and Wotawa, 2001] Markus Stumptner and Franz Wotawa. Diagnosing tree-structured systems. *Artificial Intelligence,* 127(1): 1-29,2001.