# New Structural Decomposition Techniques for Constraint Satisfaction Problems

Yaling Zheng and Berthe Y. Choueiry

Constraint Systems Laboratory
University of Nebraska-Lincoln
Email: yzheng|choueiry@cse.unl.edu

**Abstract.** We propose four new structural decomposition techniques for Constraint Satisfaction Problems. We compare these four techniques both theoretically and experimentally with hinge decomposition and hypertree decomposition. Our experiments show that one of our techniques offers the best trade-off between the computational cost of the decomposition and the width of the resulting decomposition tree.

## 1 Introduction

Many important practical problems such as scheduling, resource allocation, and product configuration can be modeled as a Constraint Satisfaction Problem (CSP), which consists of a set of variables, the domains of these variables, and a set of constraints over these variables restricting allowed combinations of values for variables. Although CSPs are in **NP**-complete in general, decomposition techniques borrowed from the area of databases have been used to characterize tractable classes of CSPs [1–4]. The basic principle is to decompose the CSP into sub-problems that are organized in a tree structure. The subproblems are then solved independently, and the solutions are propagated in a backtrack-free manner along the tree [5] to yield a solution to the initial CSP, as described by Dechter and Pearl [1]. We propose new decomposition techniques and position them in the context of the hierarchy specified by Gottlob et al. [4], which unifies main decomposition strategies and compares them in terms of generality. The main techniques are biconnected decomposition (BICOMP) [6], hinge decomposition (HINGE) [2, 3], tree clustering (TCLUSTER) [1], hinge decomposition combined with tree clustering (HINGE$^{\text{TCLUSTER}}$) [2], and hypertree decomposition (HYPERTREE) [7]. These techniques can be further characterized by their computational complexity and the width of the tree they generate (which is the size of the largest sub-problem in the tree). Among the above methods, HYPERTREE is the most general and yields trees with the smallest possible width. However, it remains costly in practice even though its complexity is polynomial [8] (see experiments in Section 8). HINGE is a more efficient but less general strategy than HYPERTREE. In this paper, we generalize HINGE into HINGE$^{+}$, and introduce CUT as a variation of HINGE. Further, we propose a new technique, TRAVERSE, which we combine with CUT to yield a new technique CaT. In summary, HINGE$^+$ generalizes HINGE, and CaT generalizes CUT. We evaluate our new techniques theoretically and empirically on randomly generated hypergraphs. Our experiments show that CaT provides the best trade-off between the width of the generated tree and the computational cost of the decomposition.

This paper is organized as follows. Section 2 reviews the preliminaries of CSPs. Section 3 introduces HINGE$^+$. Section 4 describes CUT, which is a variation of HINGE$^+$. Section 5 introduces a new technique called TRAVERSE. Section 6 combines CUT and TRAVERSE into CaT. Section 7 establishes the formal relationships among these techniques, and also with respect to HINGE and HYPERTREE. Section 8 demonstrates the effectiveness of CaT on randomly generated problems. Finally, Section 9 concludes the paper.

## 2 Background

A CSP is defined as a tuple $\mathcal{P} = (\mathcal{V}, \mathcal{D}, \mathcal{C})$, where $\mathcal{V}$ is a set of variables, $\mathcal{D}$ is a set of value domains for the variables, and $\mathcal{C}$ is a set of constraints that restrict the acceptable combination of values to variables. Every constraint $C_i \in \mathcal{C}$ is a relation over a set $S_i \subseteq \mathcal{V}$ of variables, and specifies the set of allowed tuples as a subset of the Cartesian product of the domains of $S_i$. We denote the set of variables involved in constraint $C_i$ by $\textsc{Scope}(C_i)$, and the union of the scopes of a set of constraints $\{C_i\}$ by $\textsc{Var}(\{C_i\})$. A solution to the CSP is an assignment of values to all variables such that all the constraints are simultaneously satisfied. The CSP can be represented by its associated constraint hypergraph. The constraint hypergraph of a CSP $\mathcal{P} = (\mathcal{V}, \mathcal{D}, \mathcal{C})$ is given by $\mathcal{H} = (\mathcal{V}, \mathcal{S})$, where $\mathcal{S}$ is a set of hyperedges corresponding to the scopes of the constraints in the CSP. Figure 1 shows the hypergraph $\mathcal{H}_{cg}$ of a CSP with 22 variables and 16 constraints. The primal graph of a constraint hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{S})$ is a graph

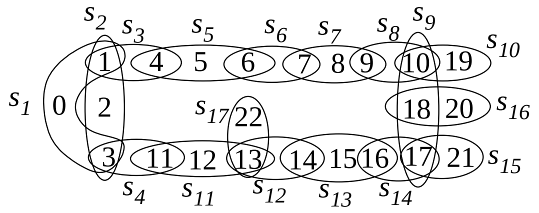

**Fig. 1.** A constraint hypergraph $\mathcal{H}_{cg}$.

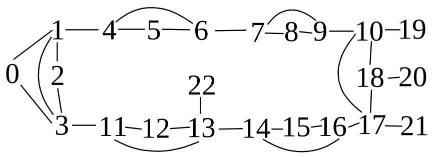

**Fig. 2.** The primal graph of $\mathcal{H}_{cg}$.

$G = (\mathcal{V}, E)$, where $E$ is a set of edges relating any 2 variables that appear in the scope of a constraint in the CSP. Figure 2 shows the primal graph of $\mathcal{H}_{cg}$. Further, we say that a hypergraph is connected when its corresponding primal graph is connected. Each connected component of the primal graph defines a connected component of the hypergraph.

*Acyclic* CSPs are those CSPs whose associated constraint hypergraph is acyclic. A constraint hypergraph $\mathcal{H}$ is acyclic iff its primal graph $G$ is chordal (i.e., every cycle of length at least 4 has an edge connecting 2 non-adjacent vertices) and conformal (i.e., there is a one-to-one mapping between each maximal clique of the primal graph and the scope of the constraints) [9]. The constraint hypergraph $\mathcal{H}_{cg}$ shown in Figure 1 is not acyclic.

Following [10], a *join tree* $JT(\mathcal{H})$ for a constraint hypergraph $\mathcal{H}$ is a tree whose nodes are the edges of $\mathcal{H}$ such that whenever the same vertex $X \in \mathcal{V}$ appears in 2 hyperedges $s_1$ and $s_2 \in \mathcal{S}$, then $s_1$ and $s_2$ are connected, and $X$ appears in each node on the unique path linking $s_1$ and $s_2$ in $JT(\mathcal{H})$. In other words, the set of nodes in which $X$ appears includes a (connected) subtree of $JT(\mathcal{H})$. The *width* $d$ of a join tree is the maximum number of hyperedges in all the nodes of the join tree. Figure 3 shows a join tree of $\mathcal{H}_{cg}$ of width $d$=2. The

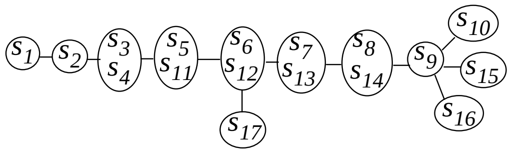

**Fig. 3.** A join tree of $\mathcal{H}_{cg}$.

principle of *structural decomposition techniques* is to compute an equivalent join tree for a given constraint hypergraph. Each node in this tree is a sub-problem for which we find all solutions, then, while applying directional arc-consistency to the join tree, we can solve the CSP in a backtrack-free manner [1, 2]. The complexity of solving the sub-problems is $O(|\mathcal{S}|l^d d \log l)$, where $l$ is the maximum size of a constraint in $\mathcal{S}$ and $d$ the width of the join tree [2]. Gottlob et al. [4] defined a set of criteria for comparing decomposition methods, where $C(D_i, k)$ is a class of CSPs for which there exists a decomposition of width $\leq k$ by the decomposition method $D_i$ that can be solved in polynomial time. These criteria are as follows (taken verbatim from [4]):

1. *Generalization.* $D_2$ generalizes $D_1$ if there exists a constant $\delta \geq 0$ such that, for each level $k$, $C(D_1, k) \subseteq C(D_2, k+\delta)$ holds. In practical terms, this means that whenever a class $C$ of constraints is tractable according to method $D_1$, it is also tractable according to $D_2$.
2. *Beating.* $D_2$ beats $D_1$ if there exists an integer $k$ such that $C(D_2, k) \not\subseteq C(D_1, m)$ for any $m$. Intuitively, this means that some classes of problems are tractable according to $D_2$ but not according to $D_1$.
3. *Strong generalization.* $D_2$ strongly generalizes $D_1$ if $D_2$ generalizes $D_1$ and $D_2$ beats $D_1$. This means that $D_2$ is really the more powerful method given that, whenever $D_1$ guarantees polynomial runtime for constraint solving, then $D_2$ also guarantees tractable constraint solving. However, there are classes of constraints that can be solved in polynomial time by using $D_2$ but are not tractable according to $D_1$.
4. *Strongly incomparable.* $D_1$ and $D_2$ are strongly incomparable if both $D_1$ beats $D_2$ and $D_2$ beats $D_1$.

Figure 4 shows the hierarchy developed by Gottlob et al. [4] based on the above comparision criteria. Whenever two decomposition methods are not related by a directed path, they are strongly incomparable.

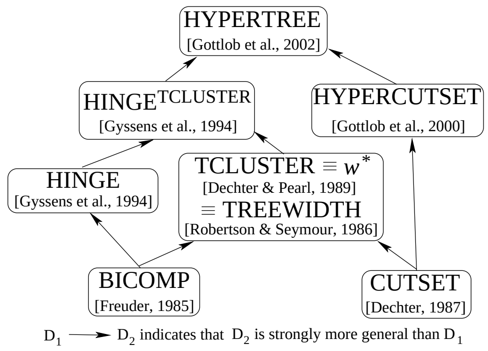

**Fig. 4.** The hierarchy of constraint tractability of [4].

## 3 Hinge$^+$ decomposition (HINGE$^+$)

In this section, we introduce HINGE$^+$ as an improvement of HINGE. As specified by Gyssens et al. [2], HINGE decomposes the constraint hypergraph into a join tree where each node (called 1-hinge) is a set of hyperedges and 2 nodes that are adjacent in the tree share exactly one hyperedge. Figure 5 shows a decomposition of $\mathcal{H}_{cg}$ of Figure 1 by HINGE where $d = 12$. The resulting decomposition guarantees a set of properties (i.e., inheritance, decomposition, and inseparability) that they define. They also attempted to generalize their approach to $k$-hinges, where a $k$-hinge is a node in the join tree connected to other nodes with at most $k$ hyperedges. However, they showed that their algorithm for 1-hinge cannot be generalized to achieve a correct result. The width of the join

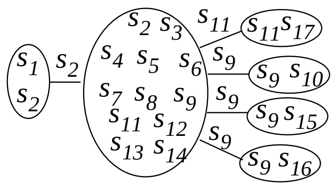

**Fig. 5.** Applying HINGE to $\mathcal{H}_{cg}$.

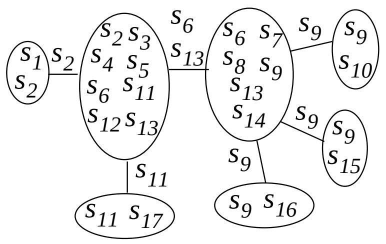

**Fig. 6.** A finer decomposition than that of Figure 5.

tree of Figure 5 is particularly high. We noticed that by allowing the nodes of the tree to connect through more than 1 hyperedge (as suggested by $k$-hinge of Jeavons et al. [3]), we can obtain a finer decomposition such as the one shown in Figure 6. We introduce 3 important definitions, which we will use to define HINGE$^+$, our improvment on HINGE:

**Definition 1.** REMAIN-HG*($F, \mathcal{S}$). Given a connected constraint hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{S})$ and a set of hyperedges $F \subseteq \mathcal{S}$, we define $\mathcal{H}_r = (\mathcal{V}_r, \mathcal{S}_r)$, denoted* REMAIN-HG*($F, \mathcal{S}$), as the remaining constraint hypergraph obtained after removing $F$ from $\mathcal{S}$. More formally: $\mathcal{V}_r = \mathcal{V} \setminus \text{VAR}(F)$ and $\mathcal{S}_r = \bigcup_{h \in \mathcal{S}} h \setminus \text{VAR}(F)$.*

**Definition 2.** *$i$-cut. Given a connected constraint hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{S})$ where $|\mathcal{S}| \geq i + 1$, an $i$-*cut *of $\mathcal{H}$ is a set of hyperedges $F$ such that:*

1. *$F \subset \mathcal{S}$ and $|F| = i$; and*
2. Remain-hg*($F$, $\mathcal{S}$) has at least 2 components.*

**Definition 3.** Max-Size*($F$, $\mathcal{H}$). Given an $i$-cut $F$ of a constraint hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{S})$,* Max-Size*($F$, $\mathcal{H}$) is the largest number of hyperedges in a connected component in* Remain-hg*($F$,$\mathcal{H}$).*

Given a constraint hypergraph $\mathcal{H}$, HINGE continuously finds 1-cuts (connecting 1-hinges). We improve HINGE by finding 1-cuts through $k$-cuts, where $k$ is a specified maximum cut-size. The difficulty here is to choose among the $i$-cuts for a given $i$ ($1< i \leq k$), as there may be more than one possible choice. We solve this problem by choosing the $i$-cut that yields the minimum value of Max-Size. Now we define the join tree resulting from HINGE$^+$:

**Definition 4.** *$k$-hinge$^+$-tree. Given a constraint hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{S})$, a $k$-hinge$^+$-tree of $\mathcal{H}$ is a tree, $T = (N, A)$, with nodes $N$ and labeled arcs $A$, such that:*

1. *For each tree node, $p \subseteq \mathcal{S}$;*
2. *For each hyperedge $h \in \mathcal{S}$, there exists a tree node $p$ such that $h \in p$;*
3. *For 2 adjacent tree nodes $p_1$ and $p_2$, there exists an $i$-cut $C$ ($1\leq i \leq k$) such that* Var*($p_1$) $\cap$* Var*($p_2$) $=$* Var*($C$); and*
4. *For each variable $Y \in \mathcal{V}$, the set $\{p \in N \mid Y \in$* Var*$(p)\}$ induces a connected subtree of $T$.*

Given a constraint hypergraph $\mathcal{H}$ and a constant number $k$, which is the maximum cut size, HINGE$^+$ (see Algorithm 1) returns a $k$-hinge$^+$-tree by finding 1-cuts through $k$-cuts. The worst case of the algorithm occurs when there are no $i$-cuts $1\leq i \leq (k-1)$. In this case, line 11 loops at most $|\mathcal{S}|^k$ times, and each loop can be performed in $O(|\mathcal{V}||\mathcal{S}|)$ time. Therefore, the worst-case time complexity of HINGE$^+$ is $O(|\mathcal{V}||\mathcal{S}|^{k+1})$. Since $k$ is used to limit the cut size, Algorithm 1 remains polynomial. Figure 7 shows a 2-hinge$^+$-tree for $\mathcal{H}_{cg}$.

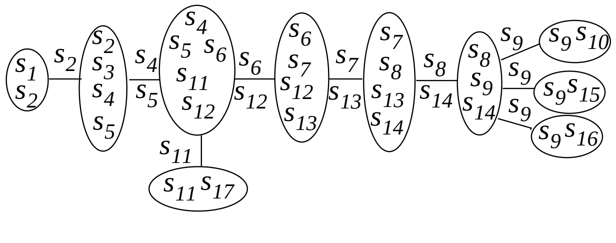

**Fig. 7.** Applying HINGE$^+$ to $\mathcal{H}_{cg}$ with $k = 2$.

```
**Input**: A hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{S})$ and a maximum cut-size $k$.
**Output**: An $k$-hinge$^+$-tree $T$ for $(\mathcal{V}, \mathcal{S})$.
1  $i \leftarrow 1$;
2  $S_{\text{cuts}} \leftarrow \emptyset$;
3  $N_i \leftarrow \{\mathcal{S}\}$;
4  Mark every hyperedge in $\mathcal{S}$ as '*unchosen*';
5  **foreach** *j from 1 to k step by 1* **do**
6      Mark the nodes in $N_i$ as *j-non-minimal*;
7      **while** *not all nodes of $N_i$ are marked j-minimal* **do**
8          Choose a *j-non-minimal* node $F$ in $N_i$;
9          *j-combinations* $\leftarrow$ all combinations of $j$ '*unchosen*' hyperedges in $F$;
10         *j-cuts* $\leftarrow \emptyset$;
11         **foreach** *j-combination $X \in$ j-combinations* **do**
12             $\Gamma \leftarrow \{G \cup X \mid G$ is a connected component in REMAIN-HG$(X, F)\}$;
13             **if** *$(|\Gamma| > 1)$ and $(\forall C_q \in \{S_{\text{cut}} \mid (S_{\text{cut}} \in S_{\text{cuts}})$ and $(S_{\text{cut}} \subseteq F)\}$,
               $\exists \Gamma_p \in \Gamma$ such that $C_q \subseteq \Gamma_p)$* **then**
14                 *j-cuts* $\leftarrow$ *j-cuts* $\cup \{X\}$;
               **end**
           **end**
15         **if** *j-cuts $\neq \emptyset$* **then**
16             choose a *j-cut* $C$ with smallest MAX-SIZE(*j-cut*, $F$);
17             Mark the hyperedges in $C$ as '*chosen*';
18             $S_{\text{cuts}} \leftarrow S_{\text{cuts}} \cup \{C\}$;
19             $\Gamma \leftarrow \{G \cup C \mid G$ is a connected component in REMAIN-HG$(C, F)\}$;
20             $N_{i+1} \leftarrow (N_i \setminus \{F\}) \cup \Gamma$;
21             Mark $C$ as a *j-cut* of every element in $\Gamma$;
22             Let $\gamma$: $\{FN_1, \ldots, FN_q\} \rightarrow \Gamma$ such that $\forall FN_i \cap \gamma(FN_i) \neq \emptyset$;
23             $A_{i+1} \leftarrow \quad (A_i \setminus \{(\{F, F'\}, C) \mid (\{F, F'\}, C) \in A_i\})$
                        $\cup\{(\{\gamma(FN), FN\}, C) \mid (\{F, FN\}, C) \in A_i\}$
                        $\cup\{(\{\Gamma_0, \Gamma_y\}, C) \mid \Gamma_0$ is an arbitrary chosen element from $\Gamma$,
                                  $\Gamma_y \in \Gamma$ and $\Gamma_y \neq \Gamma_0\}$;
24             Mark all the new nodes added to $N_{i+1}$ as *j-non-minimal*;
           **else**
25             Mark $F$ as *j-minimal*;
           **end**
26         $i \leftarrow i + 1$;
       **end**
   **end**
27 $T \leftarrow (N_i, A_i)$;
```

**Algorithm 1**: HINGE$^+$.

## 4 Cut decomposition (CUT)

In this section, we introduce CUT as a variation of HINGE$^+$. The arcs incident to every node in the equivalent join tree of a constraint hypergraph obtained by CUT are labeled by at most 2 distinct cuts. For HINGE$^+$, the arcs incident to a given node in an equivalent join tree of a constraint hypergraph obtained by HINGE$^+$ can be labeled by more than 2 distinct cuts. For example, in the join tree of Figure 7, the arcs incident to the node $\{s_4, s_5, s_6, s_{11}, s_{12}\}$ are labeled with three different cuts, namely $\{s_4, s_5\}$, $\{s_6, s_{12}\}$, and $\{s_{11}\}$. The algorithm of CUT is obtained by replacing the conditions in line 13 with the following ones:

1. $|\mathit{\Gamma}| > 1$;
2. For $\forall C_q \in \{S_{\text{cut}} \mid (S_{cut} \in S_{\text{cuts}})$ and $(S_{\text{cut}} \subseteq F)\}$, there exists $\mathit{\Gamma}_p \in \mathit{\Gamma}$ such that $C_q \subseteq \mathit{\Gamma}_p$; and
3. For every 2 sets of hyperedges $C_i$ and $C_j \in S_{\text{cuts}}$, if $C_i \neq C_j$, and $C_i \subseteq \mathit{\Gamma}_i, C_j \subseteq \mathit{\Gamma}_j$, then $\mathit{\Gamma}_i \neq \mathit{\Gamma}_j$.

The above conditions guarantee that no more than 2 cuts label the arcs incident to a node in the join tree obtained by CUT. (This feature allows us to further traverse each tree node from one cut to another cut and is exploited in Section 5.) The complexity of CUT is the same as that of HINGE$^+$. Figure 8 shows the result of applying CUT (the maximum cut size $k$ is 2) to the constraint hypergraph $\mathcal{H}_{cg}$ shown in Figure 1.

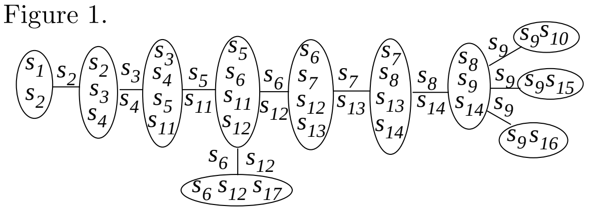

**Fig. 8.** Applying CUT to $\mathcal{H}_{cg}$.

## 5 Traverse decomposition (TRAVERSE)

In this section, we introduce a simple sweep-like decomposition technique called TRAVERSE. We describe two variations of TRAVERSE: TRAVERSE-I and TRAVERSE-II. TRAVERSE-I takes a constraint hypergraph and one set of hyperedges in it, and 'sweeps' through the hypergraph from the set of hyperedges to generate an equivalent join tree of the constraint hypergraph. TRAVERSE-II takes a constraint hypergraph and 2 sets of hyperedges from the hypergraph and 'sweeps' through the constraint hypergraph from the first set of hyperedges to the second set of hyperedges to generate an equivalent join tree of the constraint hypergraph. For convenience, we first introduce the definition of NEIGHBORS$(F, \mathcal{S})$ that will be used in Algorithm 2 and Algorithm 3.

**Definition 5.** *Neighboring hyperedges. The neighboring hyperedges of a set of hyperedges $F$ in a constraint hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{S})$ with $F \subseteq \mathcal{S}$, denoted* NEIGHBORS*$(F, \mathcal{S})$, is a set given by:*

$$\{e \mid e \in F,\ e \not\subseteq F, \text{ and } \textsc{Var}(\{e\}) \cap \textsc{Var}(F) \neq \emptyset\}. \tag{1}$$

Given a constraint hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{S})$ and a set of hyperedges $F \subseteq \mathcal{S}$, TRAVERSE-I returns a unique join tree obtained by Algorithm 2 via 'sweeping' through the constraint hypergraph starting from the hyperedges in $F$. We denote TRAVERSE-I$(\mathcal{H}, F)$ the result obtained by applying Algorithm 2 with $F$ on $\mathcal{H}$. The loop in line 7 of Algorithm 2 executes at most $|\mathcal{S}|$ times, and each

```
Input: a constraint hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{S})$ and a set of hyperedges $F \subseteq \mathcal{S}$.
Output: an equivalent join tree $T$ for $\mathcal{H}$.
1  $N \leftarrow \emptyset$;   $A \leftarrow \emptyset$;
2  Mark any hyperedge $e \in \mathcal{S}$ as 'unvisited';
3  $F_v \leftarrow \{e \mid \text{VAR}(\{e\}) \subseteq \text{VAR}(F)\}$;
4  $N \leftarrow N \cup \{F_v\}$;
5  $F_{jv} \leftarrow F_v$;
6  Mark any hyperedge in $F_{jv}$ as 'visited';
7  while not all hyperedges in $\mathcal{S}$ are 'visited' do
8      $F' \leftarrow$ NEIGHBORS($F_{jv}$, the set of all 'unvisited' hyperedges);
9      $F_v \leftarrow \{e \mid \text{VAR}(e) \subseteq \text{VAR}(F')\ \}$;
10     $N \leftarrow N \cup \{F_v\}$;
11     $A \leftarrow A \cup \{(F_{jv}, F_v)\}$;
12     $F_{jv} \leftarrow F_v$;
13     Mark every hyperedge in $F_{jv}$ as 'visited';
   end
   $T \leftarrow (N, A)$;
```

**Algorithm 2**: TRAVERSE-I.

execution can be performed in $O(|\mathcal{V}||\mathcal{S}|)$ time. Therefore, the worst-case time complexity of TRAVERSE-I is $O(|\mathcal{V}||\mathcal{S}|^2)$. Figure 9 shows the join tree computed by TRAVERSE-I starting from $\{s_1\}$ in $\mathcal{H}_{cg}$. Because it 'sweeps' through

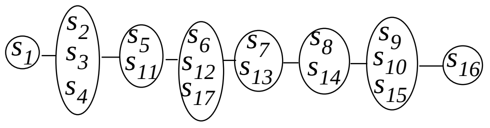

**Fig. 9.** Applying TRAVERSE-I to $\mathcal{H}_{cg}$ from $\{s_1\}$.

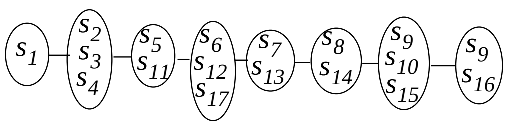

**Fig. 10.** Applying TRAVERSE-II to $\mathcal{H}_{cg}$ from $\{s_1\}$ to $\{s_9, s_{16}\}$.

the constraint hypergraph, TRAVERSE always computes a join tree that is a connected *chain*, provided the constraint hypergraph is connected. The result of the decomposition depends on $F$, the starting set of hyperedges. If we traverse $\mathcal{H}_{cg}$ of Figure 1 starting from $\{s_6, s_9, s_{12}\}$, Algorithm 2 would yield a join tree of width $d = 10$. Starting from $\{s_1\}$, the width is $d = 3$ (see Figure 9).

Our goal is to combine CUT with TRAVERSE to improve the $k$-hinge$^+$-tree computed by CUT (Section 6). To this end, we introduce TRAVERSE-II (Algorithm 3), which allows us to sweep the constraint hypergraph between 2 cuts. TRAVERSE-II takes a constraint hypergraph and 2 sets of hyperedges, and then sweeps through the constraint hypergraph from the first set of hyperedges to the second set of hyperedges to generate an equivalent join tree of this constraint hypergraph. We denote TRAVERSE-II$(\mathcal{H}, C_1, C_2)$ the result of applying

```
Input: a constraint hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{S})$, a set of hyperedges $C_1$ and another
       set of hyperedges $C_2$.
Output: an equivalent join tree $T$ for $\mathcal{H}$.
1  $N \leftarrow \emptyset$;   $A \leftarrow \emptyset$;
2  Mark any hyperedge $e \in \mathcal{S}$ as 'unvisited';
3  $F_d \leftarrow \{e \mid \textsc{Var}(e) \subseteq \textsc{Var}(C_2)\}$;
4  $F_v \leftarrow \{e \mid \textsc{Var}(e) \subseteq \textsc{Var}(C_1)\}$;
5  $N \leftarrow N \cup \{F_v\}$;
6  Mark any hyperedge in $F_{jv}$ as 'visited';
7  while ($F_v \neq F_d$) and (not all hyperedges in $\mathcal{S}$ are 'visited') do
8      $F' \leftarrow \textsc{Neighbors}(F_{jv} \setminus F_d$, the set of all 'unvisited' hyperedges $\cup F_d$ );
9      $F_v \leftarrow \{e \mid \textsc{Var}(e) \subseteq \textsc{Var}(F')\}$;
10     $N \leftarrow N \cup \{F_v\}$;
11     $A \leftarrow A \cup \{(F_{jv}, F_v)\}$;
12     $F_{jv} \leftarrow F_v$;
13     Mark every hyperedge in $F_{jv}$ as 'visited';
   end
   $T \leftarrow (N, A)$;
```

**Algorithm 3**: TRAVERSE-II.

TRAVERSE-II to $\mathcal{H}$ from $C_1$ to $C_2$. Figure 10 shows the join tree obtained by applying TRAVERSE-II to $\mathcal{H}_{cg}$ from $\{s_1\}$ to $\{s_9, s_{16}\}$. The loop in line 7 of Algorithm 3 executes at most $|\mathcal{S}|$ times, and each iteration can be performed in $O(|\mathcal{V}||\mathcal{S}|)$ time. Therefore, the complexity of TRAVERSE-II is $O(|\mathcal{V}||\mathcal{S}|^2)$.

## 6 Cut-and-Traverse decomposition (CaT)

In this section, we introduce CaT, which combines CUT with TRAVERSE. The algorithm of CaT is given in Algorithm 4. Given a constraint hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{S})$ and a maximum cut size $k$, Algorithm 4 first applies CUT to $\mathcal{H}$ and generates a $k$-hinge$^+$-tree in which the arcs incident to any tree node are labeled with at most 2 cuts. This step can be implemented in $O(|\mathcal{V}||\mathcal{S}|^{k+1})$ time. Then, Algorithm 4 applies either TRAVERSE-I or TRAVERSE-II to every tree node in the $k$-hinge$^+$-tree and generates a set of sub-join trees. Finally, the algorithm combines these sub-join trees into 1 join tree. The traverse process can be performed in $O(|V||\mathcal{S}|^2)$ time. Therefore, the complexity of CaT is $O(|\mathcal{V}||\mathcal{S}|^{k+1} + |V||\mathcal{S}|^2)$. Since $k \geq 1$, the complexity of CaT is $O(|\mathcal{V}||\mathcal{S}|^{k+1})$.

Note that the HYPERTREE algorithm computes an optimal hypertree of $\mathcal{H}$ that has a width within a given bound $d$; the algorithm returns *failure* if no such decomposition exists [10]. In CaT, the constant $k$ restricts the maximum cut size but does not restrict the width of the generated join tree. Figure 11 and Figure 12 show the equivalent join trees of $\mathcal{H}_{cg}$ computed by CaT and HYPERTREE. In this case, the widths of the join trees obtained by CaT and HYPERTREE are both equal to 2.

```
Input: A hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{S})$ and a maximum cut-size $k$.
Output: An equivalent join tree $T$ for $\mathcal{H}$.
Cut $\mathcal{H}$ into a tree with tree nodes $P_1$, ..., $P_m$ by CUT;
$N \leftarrow \emptyset$;   $A \leftarrow \emptyset$;
foreach i from 1 to m do
    switch the number of cuts labeling the arcs incident to $P_i$;
    do
        case 0
            $(N_i, A_i) \leftarrow$ TRAVERSE-I($P_i$, any hyperedge in $P_i$)
        case 1
            /* C is the only cut labeling the arc incident to $P_i$ */
            $(N_i, A_i) \leftarrow$ TRAVERSE-I($P_i$, $C$)
        case 2
            /* $C_1$ and $C_2$ are the cuts labeling the arcs incident to $P_i$ */
            if the width of TRAVERSE-II($P_i$, $C_1$, $C_2$) $\leq$ the width of
            TRAVERSE-II($P_i$, $C_2$, $C_1$) then
                $(N_i, A_i) \leftarrow$ TRAVERSE-II($P_i$, $C_1$, $C_2$)
            else
                $(N_i, A_i) \leftarrow$ TRAVERSE-II($P_i$, $C_2$, $C_1$)
            end
    end
    $N \leftarrow N \cup \{N_i\}$;
    $A \leftarrow A \cup \{A_i\}$;
end
$T \leftarrow (N, A)$;
```

**Algorithm 4**: CaT.

## 7 Characterization

In this section, we compare our 4 techniques with HINGE and HYPERTREE in terms of the criteria proposed by Gottlob et al. [4]. Then, we integrate our results into their hierarchy shown in Figure 4. Finally, we summarize the complexity of all six techniques.

First, we introduce two special classes of constraint hypergraphs borrowed from [4]: Circle($n$) (see Figure 13) and book($n$) (see Figure 14). These graphs are defined as follows. For any $n \geq 3$, *Circle(n)* is a constraint hypergraph having $n$ hyperedges $\{h_1, \ldots, h_n\}$ such that: $h_i = \{X_i, X_{i+1}\}$ for $\forall 1 \leq i \leq n-1$ and $h_n = \{X_n, X_1\}$. For any $n > 0$, *book(n)* is a constraint hypergraph with $2n+2$ vertices and $3n+1$ hyperedges that form $n$ squares (pages of the book) with exactly one common edge $\{X, Y\}$. The hyperedges are defined as follows:

- $b_0 = \{X, Y\}$;
- $b_{3i+1} = \{X, X_i\}$ for $\forall 1 \leq i \leq n$;
- $b_{3i+2} = \{X_i, Y_i\}$ for $\forall 1 \leq i \leq n$; and
- $b_{3i+3} = \{Y_i, Y\}$ for $\forall 1 \leq i \leq n$.

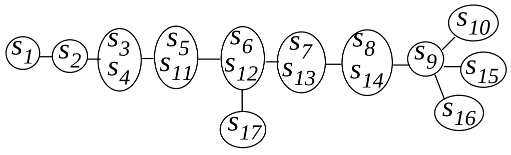

**Fig. 11.** Applying CaT to $\mathcal{H}_{cg}$.

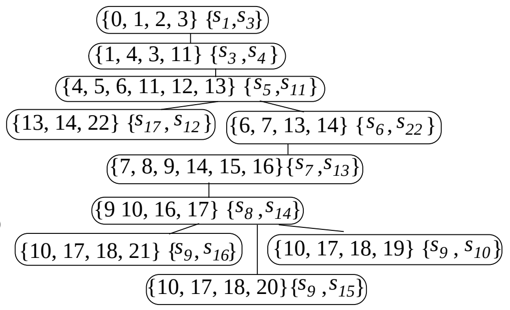
{0, 1, 2, 3} {s1,s3}
{1, 4, 3, 11} {s3 ,s4}
{4, 5, 6, 11, 12, 13} {s5 ,s11}
{13, 14, 22} {s17, s12}
{6, 7, 13, 14} {s6, s22}
{7, 8, 9, 14, 15, 16}{s7 ,s13}
{9 10, 16, 17} {s8 ,s14}
{10, 17, 18, 21} {s9, s16}
{10, 17, 18, 19} {s9 , s10}
{10, 17, 18, 20}{s9 ,s15}

**Fig. 12.** Applying HYPERTREE to $\mathcal{H}_{cg}$.

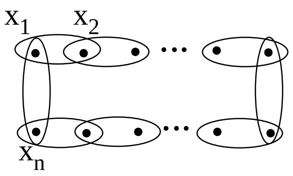

**Fig. 13.** Circle($n$).

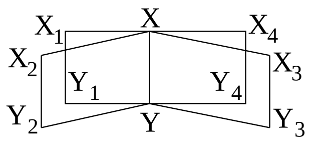

**Fig. 14.** Book(4).

**Theorem 1.** HINGE$^+$ *strongly generalizes* HINGE.

*Proof.* (HINGE$^+$ beats HINGE.) Consider the graph *Circle(n)* for some $n \geq 3$. It is easy to see that the HINGE width of *Circle(n)* is $n$, while its HINGE$^+$ width (with a maximum cut size of 2) is no greater than 4. Hence, $\bigcup_{n\geq 3}\{Circle(n)\} \subseteq C(\text{HINGE}^+, 4)$, while $\bigcup_{n\geq 3}\{Circle(n)\} \not\subseteq C(\text{HINGE}, k)$ holds for every $k > 0$. Therefore, HINGE$^+$ beats HINGE. (HINGE$^+$ generalizes HINGE.) It is easy to see that HINGE is a special case of HINGE$^+$ when the maximum cut size is 1. Thus, for $\forall I \subseteq C(\text{HINGE}, k)$, $I \subseteq C(\text{HINGE}^+, k)$ holds. □

**Theorem 2.** HYPERTREE *generalizes* HINGE$^+$.

*Proof.* It is obvious that $\forall I \subseteq C(\text{HINGE}^+, k)$, $I \subseteq C(\text{HYPERTREE}, k)$ holds. □

**Theorem 3.** CaT *generalizes* CUT.

*Proof.* The first phase of CaT is CUT. The second phase of CaT further decomposes each tree node of the join tree obtained by CUT. It is easy to see that $\forall I \subseteq C(\text{CUT}, k)$, $I \subseteq C(\text{CaT, k})$ holds. □

**Theorem 4.** HYPERTREE *generalizes* CaT.

*Proof.* It is obvious that $\forall I \subseteq C(\text{CaT}, k)$, $I \subseteq C(\text{HYPERTREE}, k)$ holds. □

**Theorem 5.** HYPERTREE *strongly generalizes* TRAVERSE.

*Proof.* (HYPERTREE generalizes TRAVERSE.) It is obvious that $\forall\, I \subseteq C$(TRAVERSE, $k$), $I \subseteq C$(HYPERTREE, $k$) holds. (HYPERTREE beats TRAVERSE.) Consider the graph *book(n)* for some $n \geq 1$, it is easy to see that the TRAVERSE width of *book(n)* is greater than $\lceil \frac{n}{2} \rceil$, while its HYPERTREE width is 2. Hence, $\bigcup_{n\geq 1}\{\mathit{book(n)}\} \subseteq C$(HYPERTREE, 2), while $\bigcup_{n\geq 1}\{\mathit{book(n)}\} \nsubseteq C$(TRAVERSE, $k$) for every $k > 0$. □

**Theorem 6.** HINGE *and* TRAVERSE *are strongly incomparable.*

*Proof.* (HINGE beats TRAVERSE.) Consider the graph *book(n)* for some $n \geq 1$, it is easy to see that the TRAVERSE width of *book(n)* is greater than $\lceil \frac{n}{2} \rceil$, while its HINGE width is 4. Hence, $\bigcup_{n\geq 1}\{\mathit{book(n)}\} \subseteq C$(HINGE$^+$, 4), while $\bigcup_{n\geq 1}\{\mathit{book(n)}\} \nsubseteq C$(HINGE, $k$) for every $k > 0$. (TRAVERSE beats HINGE.) Consider the graph *Circle(n)* for some $n \geq 3$. It is easy to see that the HINGE width of *Circle(n)* is $n$ while its TRAVERSE width (from an arbitrary chosen hyperedge) is 2. Hence, $\bigcup_{n\geq 3}\{\mathit{Circle(n)}\} \subseteq C$(TRAVERSE, 2), while $\bigcup_{n\geq 3}\{\mathit{Circle(n)}\} \nsubseteq C$(HINGE, $k$) holds for every $k > 0$. Therefore, TRAVERSE beats HINGE. □

**Theorem 7.** CUT *beats* TRAVERSE.

*Proof.* Consider the graph *book(n)* for some $n \geq 1$, it is easy to see that the TRAVERSE width of *book(n)* is greater than $\lceil \frac{n}{2} \rceil$, while its CUT width is 4. Hence, $\bigcup_{n\geq 1}\{\mathit{book(n)}\} \subseteq C$(CUT, 4), while $\bigcup_{n\geq 1}\{\mathit{book(n)}\} \nsubseteq C$(TRAVERSE, $k$) for every $k > 0$. □

**Theorem 8.** CaT *beats* TRAVERSE.

*Proof.* Consider the graph *book(n)* for some $n \geq 1$, It is easy to see that the TRAVERSE width of *book(n)* is greater than $\lceil \frac{n}{2} \rceil$ while its CaT width (with the maximum cut size being 2) is 2. Hence, $\bigcup_{n\geq 1}\{\mathit{book(n)}\} \subseteq C$(CaT, 2), while $\bigcup_{n\geq 1}\{\mathit{book(n)}\} \nsubseteq C$(TRAVERSE, $k$) for every $k > 0$. □

**Theorem 9.** HINGE$^+$ *beats* TRAVERSE.

*Proof.* Consider the graph *book(n)* for some $n \geq 1$, it is easy to see that the TRAVERSE width of *book(n)* is greater than $\lceil \frac{n}{2} \rceil$, while its HINGE$^+$ width is 4. Hence, $\bigcup_{n\geq 1}\{\mathit{book(n)}\} \subseteq C$(HINGE$^+$, 4), while $\bigcup_{n\geq 1}\{\mathit{book(n)}\} \nsubseteq C$(TRAVERSE, $k$) for every $k > 0$. □

**Theorem 10.** CUT *beats* HINGE.

*Proof.* Consider the graph *Circle(n)* for some $n \geq 3$. It is easy to see that the HINGE width of *Circle(n)* is $n$, while its CUT width (with maximum cut size being 2) is 2. Hence, $\bigcup_{n\geq 3}\{\mathit{Circle(n)}\} \subseteq C$(CUT, 2), while $\bigcup_{n\geq 3}\{\mathit{Circle(n)}\}$ $\nsubseteq C$(HINGE, $k$) holds for every $k > 0$. Therefore, CUT beats HINGE. □

The above theorems implied that CaT beats HINGE and HYPERTREE generalizes CUT. The relationships between HINGE$^+$ and CUT and between HINGE$^+$ and CaT are still need to be investigated. Figure 15 summarizes the main relationships studied above. The solid directed edge from $D_1$ to $D_2$ indicates that $D_2$ strongly generalizes $D_1$. The dotted directed edge from $D_1$ to $D_2$ indicates $D_2$ generalizes $D_1$. Note that the picture is incomplete. Table 1 summarizes the complexity of the techniques shown in Figure 15.

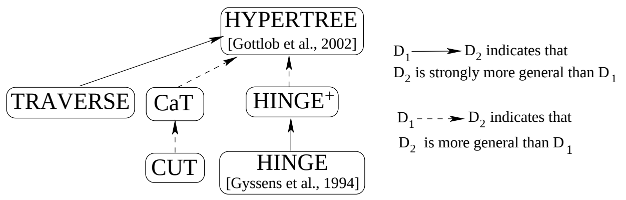
HYPERTREE
[Gottlob et al., 2002]
TRAVERSE
CaT
HINGE+
CUT
HINGE
[Gyssens et al., 1994]
D1 → D2 indicates that D2 is strongly more general than D1
D1 - - → D2 indicates that D2 is more general than D1

**Fig. 15.** Illustrating the relationships between the various studied techniques.

**Table 1.** Complexity of decomposition methods.

| **Technique** | Complexity |
|---|---|
| HYPERTREE | |
| Normal form: *opt-d-decomp* [7] | $O(\|\mathcal{S}\|^{2d}\|\mathcal{V}\|^2)$ |
| Reduced normal form [8] | Best case: $O(\|\mathcal{S}\|^d\|\mathcal{V}\| + \|\mathcal{S}\|^2\|\mathcal{V}\|)$ |
| HINGE | $O(\|\mathcal{V}\|\|\mathcal{S}\|^2)$ |
| HINGE$^+$ | $O(\|\mathcal{V}\|\|\mathcal{S}\|^{k+1})$ |
| CUT | $O(\|\mathcal{V}\|\|\mathcal{S}\|^{k+1})$ |
| TRAVERSE | $O(\|\mathcal{V}\|\|\mathcal{S}\|^2)$ |
| CaT | $O(\|\mathcal{V}\|\|\mathcal{S}\|^{k+1})$ |
| Solving the CSP after decomposition | $O(\|\mathcal{S}\|l^d d \log l)$ |

$|\mathcal{V}|$: number of variables (i.e., vertices)
$|\mathcal{S}|$: number of constraints (i.e., hyperedges)
$d$: width of the join tree resulting from a decomposition
$k$: maximum cut-size
$l$: maximum size of a constraint in $\mathcal{S}$

## 8 Preliminary experiments

In order to assess empirically the above techniques, we compared their performance on randomly generated hypergraphs in terms of two criteria: the CPU time for computing the decompositions and the width of the resulting join tree. For HYPERTREE, we used the algorithm of Harvey and Ghose [8], which improves on the opt-$k$-decomp algorithm of Gottlob et al. [10]. By starting with $k$=1 and incrementing its value by 1 until it finds decomposition, the algorithm we used guarantees an optimal decomposition. We generated random hypergraphs setting the number of constraints to 10, 11, 12, and 13. In each instance, we chose the arity of the constraints randomly in $\{2, 3, 4\}$. Table 2 summarizes the constraint hypergraphs used in the experiments. We set the maximum cut size $k$=2 for HINGE$^+$, CUT, and CaT. Figure 16 and Figure 17 show, for a fixed number of constraints, the average CPU times and average widths of the generated join trees. Figure 16 and Figure 17 show the average CPU times and average widths of different decomposition techniques. Table 3 averages these results over all 4000 instances generated.

**Table 2.** Constraint hypergraphs used in the experiments.

| # constraints | # variables | # instances |
|---|---|---|
| 10 | {16, 17, ..., 25} | 1000 (100 instances for each fixed number of variables) |
| 11 | {18, 19, ..., 27} | 1000 (100 instances for each fixed number of variables) |
| 12 | {20, 21, ..., 29} | 1000 (100 instances for each fixed number of variables) |
| 13 | {22, 23, ..., 31} | 1000 (100 instances for each fixed number of variables) |

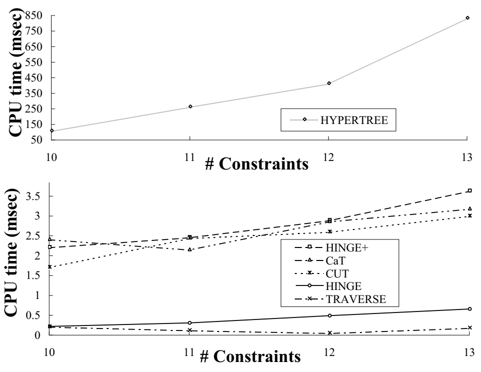

**Fig. 16.** Average CPU times.

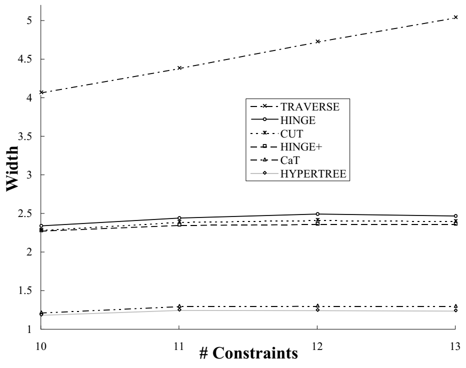

**Fig. 17.** Average widths.

**Table 3.** Average results over all 4000 instances.

| Comparison criteria Average | HINGE | HINGE$^+$ | CUT | TRAVERSE | CaT | HYPERTREE |
|---|---|---|---|---|---|---|
| CPU time [msec] | 0.400 | 2.786 | 2.428 | 0.130 | 2.640 | 400.900 |
| Width | 2.425 | 2.332 | 2.367 | 4.547 | 1.273 | 1.225 |

From these experiments, we have the following observations:

**For CPU time,**

TRAVERSE < HINGE < CUT $\approx$ CaT $\approx$ HINGE$^+$ $\ll$ HYPERTREE.
TRAVERSE is the quickest technique followed by HINGE then CaT, HINGE$^+$, and CUT, which have comparable values for the CPU time. All techniques are significantly quicker than HYPERTREE. Indeed, the computationally cost of HYPERTREE is prohibitively high although its worst-case time complexity is polynomial.

**For width,**

HYPERTREE $\approx$ CaT< HINGE$^+$ $\approx$ CUT $\approx$ HINGE < TRAVERSE.
The join tree obtained with TRAVERSE has the largest width. The average widths of the join tree generated by HINGE$^+$ and CUT are smaller than that of the join tree generated by HINGE. However, the differences of these values are within 4%. The widths of the join trees generated by CaT and HYPERTREE differ by only 4%, which is negligible. Also, they are significantly smaller than those generated by the remaining techniques.

In summary, CaT offers the best trade-off between the CPU time and the width of the computed join tree among the decomposition methods tested.

## 9 Conclusion

In this paper, we proposed two main new structural decompositions: HINGE$^+$ and CaT. HINGE$^+$ strongly generalizes HINGE of Gyssens et al. [2]. CaT is built by combining CUT (a variation of HINGE$^+$) and TRAVERSE (a sweep-like decomposition techniques). We compared these techniques among themselves and with HINGE and HYPERTREE both theoretically and experimentally. Our experiments showed that the CaT offers the best trade-off between cost and quality of the resulting decomposition.

In the future, we plan to address the following issues: (1) Compare our techniques with the remaining techniques shown in Figure 4; and (2) Perform experiments on special types of graphs (e.g., small-world graphs and clustered graphs) and real-world problems (e.g., the ones used in [11]).

*Acknowledgments:* This work is supported by CAREER Award #0133568 from the National Science Foundation. The experiments were conducted utilizing the Research Computing Facility of the University of Nebraska-Lincoln. Deb Derrick provided invaluable editorial help.

## References

1. Dechter, R., Pearl, J.: Tree Clustering for Constraint Networks. Artificial Intelligence **38** (1989) 353–366
2. Gyssens, M., Jeavons, P.G., Cohen, D.A.: Decomposing Constraint Satisfaction Problems Using Database Techniques. Artificial Intelligence **66** (1994) 57–89
3. Jeavons, P.G., Cohen, D.A., Gyssens, M.: A Structural Decomposition for Hypergraphs. Contemporary Mathematics **178** (1994) 161–177
4. Gottlob, G., Leone, N., Scarcello, F.: A Comparison of Structural CSP Decomposition Methods. Artificial Intelligence **124** (2000) 243–282
5. Freuder, E.C.: A Sufficient Condition for Backtrack-Free Search. JACM **29 (1)** (1982) 24–32
6. Freuder, E.C.: A Sufficient Condition for Backtrack-Bounded Search. JACM **32 (4)** (1985) 755–761
7. Gottlob, G., Leone, N., Scarcello, F.: Hypertree Decompositions and Tractable Queries. Journal of Computer and System Sciences **64** (2002) 579–627
8. Harvey, P., Ghose, A.: Reducing Redundancy in the Hypertree Decomposition Scheme. In: The 15$^{th}$ IEEE International Conference on Tools with Artificial Intelligence (ICTAI 03). (2003) 474–481
9. Dechter, R.: Constraint Processing. Morgan Kaufmann (2003)
10. Gottlob, G., Leone, N., Scarcello, F.: On Tractable Queries and Constraints. In: 10$^{th}$ International Conference and Workshop on Database and Expert System Applications (DEXA 1999). (1999) 1–15
11. Gottlob, G., Hutle, M., Wotawa, F.: Combining Hypertree, Bicomp, And Hinge Decomposition. In: Proc. of the 15 $^{th}$ ECAI, Lyon, France (2002) 161–165