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. Techniques for decomposing the constraint network of a Constraint Satisfaction Problem (CSP) have yielded the identification of new tractable classes of CSPs [1]. In this paper, we propose three new decomposition techniques, namely HINGE$^+$, CUT, and TRAVERSE, where HINGE$^+$ is a generalization of HINGE by Gyssens et al. [2]. Further, we combine CUT and TRAVERSE into a new strategy, which we call Cut-and-Traverse (CaT). We introduce these techniques and compare them according to the criteria introduced by Gottlob et al. [1].

1 Introduction

Many important practical problems such as scheduling, resource allocation, design and product configuration can be modeled as a Constraint Satisfaction Problem (CSP), where a set of decisions need to be made, each decision has a number of options, and the allowable combinations of these options are restricted by a set of constraints. Because a CSP is in general **NP**-complete, search remains the ultimate mechanism for solving it. Decomposition is a common strategy for improving the performance of search [3]. Recently, decomposition techniques borrowed from the area of databases (i.e., acyclic conjunctive queries) have been used to characterize tractable classes of CSPs [4, 2, 5, 1]. 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 [6] to yield a solution to the initial CSP. We propose new decomposition techniques and we position them in the context of the hierarchy specified by Gottlob et al. [1], which unifies major decomposition strategies reported in the literature and compares them in terms of generality. The main techniques are the biconnected decomposition (BICOMP) [6], hinge decomposition (HINGE) [2, 5], tree clustering (TCLUSTER) [4], hinge decomposition combined with tree clustering (HINGE$^{\text{TCLUSTER}}$) [2], and hypertree decomposition (HYPERTREE) [7]. These techniques can be further characterized by the width of the tree they generate and the their computational complexity. Among the above methods, HYPERTREE is the most general and yields trees with the smallest possible width. However, its cost is high. 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.

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. Finally, Section 8 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 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. To solve a CSP we need to determine whether the CSP has a solution and, if so, find one solution. Although CSPs are **NP**-complete in general, some of the CSPs are tractable due to their restricted structure. The CSP can be represented with a constraint network, which is, in general, a 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

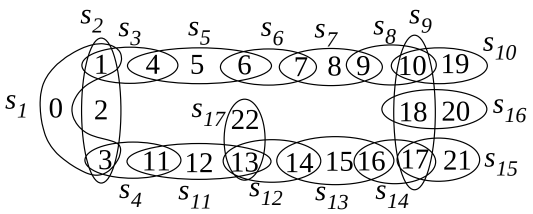

Fig. 1. A constraint hypergraph $\mathcal{H}_{cg}$.

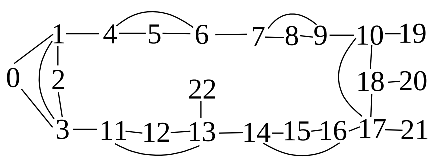

Fig. 2. The primal graph of $\mathcal{H}_{cg}$.

primal graph of a constraint hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{S})$ is a graph $G = (\mathcal{V}, E)$ where E is a set of edges relating any two 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 graphs 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., any cycle of length greater than 3 has a chord) 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) [8]. $\mathcal{H}_{cg}$ of Figure 1 is not acyclic.

Following [9], 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 two 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.

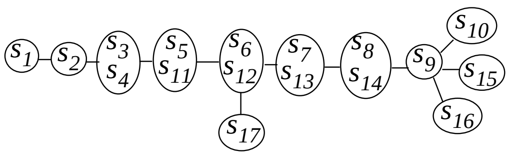

Fig. 3. A join tree of $\mathcal{H}_{cg}$.

A *structural decomposition* technique computes 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. 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. Figure 4 shows the hierarchy, proposed by Gottlob et al. of known decomposition techniques [1]. This hierarchy

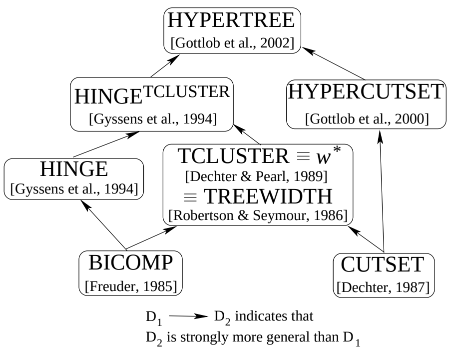

Fig. 4. The hierarchy of constraint tractability of [1].

characterized the relationships between these decomposition techniques in terms of the following criteria, 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 [1][1]:

1. *Generalization.* D_2 generalizes D_1 if there exists a constant δ such that, for each level k, $C(D_1, k) \subseteq C(D_2, k)$ 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 $C(D_2, k)$ is not contained in class $C(D_1, m)$ 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.

3 Hinge$^+$ decomposition (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 two 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 tree of Figure 5 is particularly high. We noticed that by allowing the nodes of the tree to connect through more than one hyperedge (as suggested by k-hinge of Jeavons et al. [5]), we can obtain better decompositions such as the one we show in Figure 6. We introduce the concept of k-cut to achieve such a result, which yields HINGE$^+$, our improvment on HINGE.

Consider a 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 $\textsc{Remain-hg}(F, \mathcal{H})$, as the remaining constraint hypergraph obtained after removing F from $\mathcal{H}$. More formally:

$$\mathcal{V}_r = \mathcal{V} \setminus \textsc{Var}(F)$$
$$\mathcal{S}_r = \bigcup_{h \in \mathcal{S}} h \setminus \textsc{Var}(F).$$

[1] In this hierarchy methods that are not related are guaranteed not comparable.

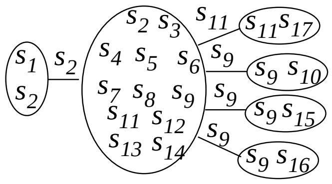

Fig. 5. Applying HINGE to $\mathcal{H}_{cg}$.

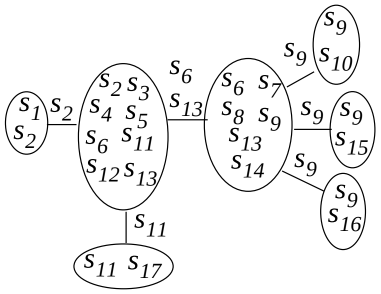

Fig. 6. A better decomposition than that of Figure 5.

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

1. *$F \subset \mathcal{S}$ and $|F| = i$; and*
2. *The remaining constraint hypergraph has at least 2 components.*

Definition 2 (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*($F, \mathcal{H}$).*

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

Given a constraint hypergraph $\mathcal{H}$ and a constant number k, the hinge$^+$ decomposition algorithm shown in Figure 1 returns a k-hinge$^+$-tree by finding 1-cuts through k-cuts. In Algorithm 1, k is the maximum cut size. The worst case of the algorithm occurs when there are no i-cuts $1 \leq i \leq (k-1)$. In the worst case,

Input: A hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{S})$ and a maximum cut-size k.

Output: An k-hinge^{+}-tree T for $(\mathcal{V}, \mathcal{S})$.

$i \leftarrow 1$;
$S_{\text{cuts}} \leftarrow \emptyset$;
$N_i \leftarrow \{\mathcal{S}\}$;
Mark every hyperedge in $\mathcal{S}$ as '*unchosen*';
foreach *j from 1 to k step by 1* **do**
    Mark the nodes in N_i as *j-non-minimal*;
    while *not all nodes of N_i are marked j-minimal* **do**
        Choose a *j-non-minimal* node F in N_i;
        j-combinations $\leftarrow$ all combinations of j '*unchosen*' hyperedges in F;
        j-cuts $\leftarrow \emptyset$;
        foreach *j-combination $X \in j$-combinations* **do**
            $\mathit{\Gamma} \leftarrow \{G \cup X \mid G$ is a connected component in REMAIN-HG$(X, F)\}$;
            if *$(|\mathit{\Gamma}| > 1)$ and $(\forall\ C_q \subseteq S_{\text{cuts}}, \exists \mathit{\Gamma}_p \in \mathit{\Gamma}$ such that $C_q \subseteq \mathit{\Gamma}_p)$;* **then**
                j-cuts $\leftarrow$ *j-cuts* $\cup \{X\}$;
            end
        end
        if *j-cuts* $\neq \emptyset$ **then**
            choose a *j-cut* C with smallest MAX-SIZE(*j-cut*, F);
            Mark the hyperedges in C as '*chosen*';
            $S_{\text{cuts}} \leftarrow S_{\text{cuts}} \cup \{C\}$;
            $\mathit{\Gamma} \leftarrow \{G \cup C \mid G$ is a connected component in REMAIN-HG$(C, F)\}$;
            $N_{i+1} \leftarrow (N_i \setminus \{F\}) \cup \mathit{\Gamma}$;
            Mark C as a *j-cut* of every element in $\mathit{\Gamma}$;
            Let γ: $\{FN_1, \ldots, FN_q\} \rightarrow \mathit{\Gamma}$ such that $\forall FN_i \cap \gamma(FN_i) \neq \emptyset$;
            $A_{i+1} \leftarrow \ (A_i \setminus \{(\{F, F'\}, C) \mid (\{F, F'\}, C) \in A_i\})$
                        $\cup\{(\{\gamma(FN), FN\}, C) \mid (\{F, FN\}, C) \in A_i\}$
                        $\cup\{(\{\mathit{\Gamma}_0, \mathit{\Gamma}_y\}, C) \mid \mathit{\Gamma}_0$ is an arbitrary chosen element from $\mathit{\Gamma}$,
                                $\mathit{\Gamma}_y \in \mathit{\Gamma}$ and $\mathit{\Gamma}_y \neq \mathit{\Gamma}_0\}$;
            Mark all the new nodes added to N_{i+1} as *j-non-minimal*;
        else
            Mark F as *j-minimal*;
        end
        $i \leftarrow i + 1$;
    end
end
$T \leftarrow (N_i, A_i)$;

Algorithm 1: Algorithm of HINGE^{+}.

line 11 loops at most $|\mathcal{S}|^k$ times, and each loop can be performed in $O(|\mathcal{V}||(\mathcal{S}|)$ time. So the worst 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 6 shows a 2-hinge$^+$-tree for $\mathcal{H}_{cg}$.

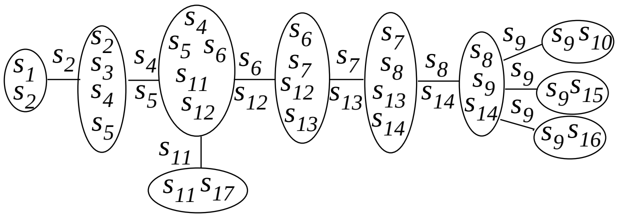

Fig. 7. Applying HINGE$^+$ on $\mathcal{H}_{cg}$ with $k = 2$.

4 Cut decomposition (CUT)

Notice that, in general, the arcs incident to a given node in a join tree may be labeled by two or more 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 $\{s_{11}\}$. In this section, we consider a variation of HINGE$^+$ called CUT, which restricts to at most 2 the number of different cuts labeling the arcs incident to any given node in the join tree. This is achieved by replacing the conditions in line 13 with the following ones:

1. $|\Gamma| > 1$;
2. $\forall\ C_q \in S_{\text{cuts}}$, there exists $\Gamma_p \in \Gamma$ such that $C_q \subseteq \Gamma_p$; and
3. For every two sets of hyperedges C_i and $C_j \in S_{\text{cuts}}$, if $C_i \neq C_j$, and $C_i \subseteq \Gamma_i, C_j \subseteq \Gamma_j$, then $\Gamma_i \neq \Gamma_j$.

The above conditions guarantee that no more than two 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$^+$, and it thus is polynomial. Figure 8 shows the result of applying CUT to the constraint hypergraph $\mathcal{H}_{cg}$ of Figure 1 with k =2.

5 Traverse decomposition (TRAVERSE)

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

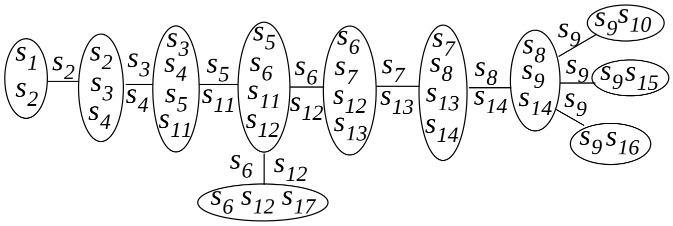

Fig. 8. Applying CUT on $\mathcal{H}_{cg}$.

Definition 4 (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 $\textsc{Neighbors}(F, \mathcal{S})$, is a set given by:

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

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 ← ∅;   A ← ∅;
2  Mark any hyperedge e ∈ S as 'unvisited';
3  F_v ← {e | Var({e}) ⊆ Var(F)};
4  N ← N ∪ {F_v};
5  F_jv ← F_v;
6  Mark any hyperedge in F_jv as 'visited';
7  while not all hyperedges in S are 'visited' do
8      F' ← Neighbors(F_jv, the set of all 'unvisited' hyperedges);
9      F_v ← {e | Var(e) ⊆ Var(F') };
10     N ← N ∪ {F_v};
11     A ← A ∪ {(F_jv, F_v)};
12     F_jv ← F_v;
13     Mark every hyperedge in F_jv as 'visited';
   end
   T ← (N, A);
```

Algorithm 2: Algorithm for TRAVERSE-I.

The loop in line 7 of the Algorithm 2 executes at most $|\mathcal{S}|$ times, and each execution can be performed in $O(|\mathcal{V}||(\mathcal{S}|)$ time. Therefore, the complexity TRAVERSE is $|\mathcal{V}||\mathcal{S}|^2)$ and is polynomial. Figure 9 shows the join tree computed by traverse decomposition from $\{s_1\}$ in $\mathcal{H}_{cg}$.

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 yields a join tree of width $d = 10$. Starting from $\{s_1\}$, the width is $d = 3$ (see Figure 9).

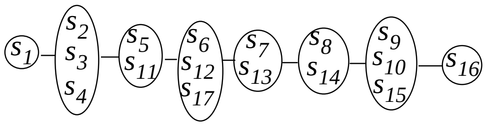

Fig. 9. Applying TRAVERSE on $\mathcal{H}_{cg}$.

We can combine CUT with TRAVERSE to improve the k-hinge$^+$-tree computed by CUT. To this end, we need to modify the Algorithm 2 in order to allow it to sweep the constraint hypergraph between two cuts. In order to traverse a constraint hypergraph from one set of hyperedges to another set of hyperedges, which will be used in Section 6, we only need to revise Algorithm 2 to Algorithm 3. We denote by TRAVERSE-II$(\mathcal{H}, C_1, C_2)$ the result of applying Algorithm 3 C_1 to C_2 on $\mathcal{H}$ The complexity of Algorithm 3 is also $O(|\mathcal{V}||\mathcal{S}|^2)$.

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 $\mathcal{H}$.
$N \leftarrow \emptyset$; $A \leftarrow \emptyset$;
Mark any hyperedge $e \in \mathcal{S}$ as '*unvisited*';
$F_d \leftarrow \{e \mid \text{VAR}(e) \subseteq \text{VAR}(C_2)\}$;
$F_v \leftarrow \{e \mid \text{VAR}(e) \subseteq \text{VAR}(C_1)\}$;
$N \leftarrow N \cup \{F_v\}$;
Mark any hyperedge in F_{jv} as '*visited*';
while *($F_v \neq F_d$) and (not all hyperedges in $\mathcal{S}$ are* 'visited'*)* **do**
  $F' \leftarrow$ NEIGHBORS$(F_{jv} \setminus F_d$, the set of all '*unvisited*' hyperedges $\cup F_d$);
  $F_v \leftarrow \{e \mid \text{VAR}(e) \subseteq \text{VAR}(F')\}$;
  $N \leftarrow N \cup \{F_v\}$;
  $A \leftarrow A \cup \{(F_{jv}, F_v)\}$;
  $F_{jv} \leftarrow F_v$;
  Mark every hyperedge in F_{jv} as '*visited*';
end
$T \leftarrow (N, A)$;

Algorithm 3: Algorithm for TRAVERSE-II.

6 Cut-and-Traverse decomposition (CaT)

In this section, we introduce CaT, which combines CUT with TRAVERSE. The algorithm for CaT is given in Algorithm 4.

Theorem 1. *Given a constraint hypergraph $\mathcal{H}$ and a constant number k, the CaT algorithm computes an equivalent join tree for $\mathcal{H}$.*

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*
      $(N_i, A_i) \leftarrow$ TRAVERSE-I(P_i, C) where C is the only cut labeling the arcs incident to P_i
    ;
    case *2*
      $(N_i, A_i) \leftarrow$ TRAVERSE-II(P_i, C_1, C_2) where C_1 and C_2 are the cuts labeling the arcs incident to P_i
    ;
  end
  $N \leftarrow N \cup \{N_i\}$;
  $A \leftarrow A \cup \{A_i\}$;
end
$T \leftarrow (N, A)$;

Algorithm 4: The algorithm for CaT.

Proof. Algorithm 4 first applies CUT to computes T_1 a k-hinge$^+$-tree equivalent to a constraint hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{S})$. The arcs incident to any tree node in T_1 are labeled with at most two cuts. The algorithm traverses each tree node in T_1 by applying TRAVERSE. If there is only one tree node in T_1, then CaT becomes TRAVERSE, which exactly computes an equivalent join tree for $\mathcal{H}$. If there are at most 2 tree nodes in T_1, then each tree node contains at least 1 cut and at most 2 cuts. Since the result of TRAVERSE on each tree node containing 1 cut is a chain-like join tree that begins with one cut, the result of TRAVERSE on each tree node containing 2 cuts is also a chain-like join tree that begins with one cut and ends with another cut. The sub join trees for all tree nodes are exactly connected through these cuts, which guarantees the connectedness property of the combined join tree of these sub join trees. Thus, the CaT algorithm computes an equivalent join tree for $\mathcal{H}$. □

HYPERTREE [9] computes an optimal hypertree of $\mathcal{H}$ with a width within a given bound k; the algorithm returns *failure* if no such decomposition exists. In our approach, the constant k only restricts the size of cuts; it does not restrict the width of the join tree computed by CaT. Therefore, CaT is more flexible than HYPERTREE.

Given a constraint hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{S})$ and a constant number k, CaT first computes an k-hinge$^+$-tree T_1 by CUT, which can be implemented in $O(|\mathcal{V}||(\mathcal{S}|^{k+1})$. The traverse process can be performed in $O(|V||(\mathcal{S}|^2)$. 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 $O(|\mathcal{V}||\mathcal{S}|^{k+1})$.

Figure 10 and Figure 11 show the equivalent trees of $\mathcal{H}_{cg}$ computed by CaT and HYPERTREE. In this case, the width of the decompositions obtained by CaT and HYPERTREE are the same and equal to 2.

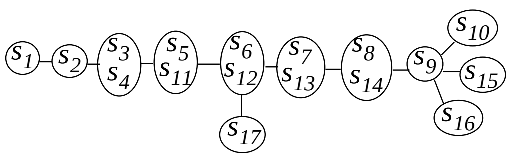

Fig. 10. Applying CaT on $\mathcal{H}_{cg}$.

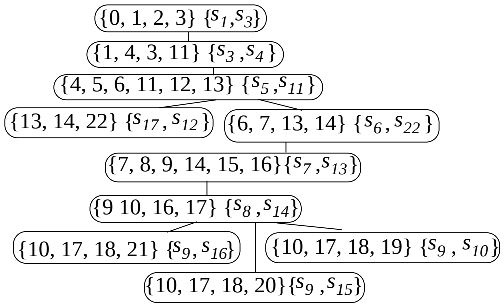

Fig. 11. Applying HYPERTREE on $\mathcal{H}_{cg}$.

7 Characterization

In this section, we compare HINGE, HINGE$^+$, CUT, TRAVERSE, CaT and HYPERTREE in terms of the criteria proposed by Gottlob et al. in [1]. Finally, we enrich the new constraint tractability hierarchy proposed by these researchers.

Theorem 2. *CaT strongly generalizes CUT.*

Proof. The first step of CaT is CUT. The second step of CaT is TRAVERSE. TRAVERSE will improve or keep the same decomposition result by CUT, thus CaT generalizes CUT. For H_{cg} shown in Figure 1, CaT computes a join tree with width 2 when limiting cut size to 2, as shown in Figure 10. CUT computes a join tree with width 4 when limiting cut size to 2, as shown in Figure 8. Thus, CaT beats CUT. Therefore, CaT strongly generalizes CUT. □

Theorem 3. *HINGE$^+$ strongly generalizes HINGE.*

Proof. When $k = 1$, HINGE$^+$ is exactly the same as HINGE. When $k > 1$, HINGE$^+$ first performs HINGE, then continuously finds 2-cuts through k-cuts, which will improve or keep the same decomposition result by HINGE. Thus, HINGE$^+$ generalizes HINGE. For H_{cg} shown in Figure 1, HINGE$^+$ computes a join tree with width 5 when limiting cut size to 2, as shown in Figure 7, while HINGE computes a join tree with width 12, as shown in Figure 5. Thus, CaT beats CUT. Therefore, HINGE$^+$ strongly generalizes HINGE. □

Also, Since HYPERTREE always compute an optimal decomposition, while HINGE$^+$, CUT and CaT are based on heuristic function, HYPERTREE generalizes these methods. For TRAVERSE, the decomposition result depends on the set of hyperedges it begins with, thus it cannot compare with HINGE, HINGE$^+$, CUT, and CaT.

Gottlob et al. [1] introduces the following comparison criteria to compare 2 different decomposition methods D_1 and D_2.

To compare HINGE, HINGE$^+$, CUT, TRAVERSE, CaT and HYPERTREE with each other, we give two additional constraint hypergraphs borrowed from [1].

For any $n > 0$, let *triangles(n)* be the graph (V, E) define as follows. The set of vertices V contains $2n + 1$ vertices $p_1, \dots, p_{2n+1}$. For each even index $i, 2 \leq i \leq 2n$, $\{p_i, p_{i-1}\}$, $\{p_i, p_{i+1}\}$, and $\{p_{i-1}, p_{i+1}\}$ are edges in E. No other edges belong to E. Figure 12 shows the $triangle(3)$. The hypertree width of $triangles(\mathrm{n})$ is 2. In fact, a hypertree (T, χ, λ), where T is a simple chain of n vertices $v_1, \dots, v_n$, and, for each $v_i (1 \leq i \leq n)$, $\chi(v_i) = \{p_{2i-1}, p_{2i}, p_{2i+1}\}$ and $\lambda(v_i)$ contains the two edges $\{p_{2i-1}, p_{2i}\}$ and $\{p_{2i}, p_{2i+1}\}$, is a width 2 hypertree decomposition of *triangles(n)*.

For any $n > 0$, let *book(n)* be a graph having $2n + 2$ vertices and $3n + 1$ edges that form n squares (pages of the book) having exactly one common edge $\{X, Y\}$. It is easy to see that the hypertree width of *book(n)* is 2. Figure 13 shows the graph *book(4)*.

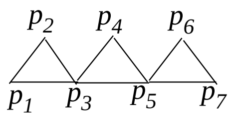

Fig. 12. Triangle(3).

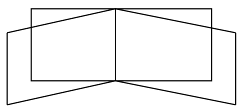

Fig. 13. Book(4).

Theorem 4. *HYPERTREE strongly generalizes CaT.*

Proof. HYPERTREE computes a join tree with the smallest possible width for a constraint hypergraph. Because CaT a heuristic decomposition method, it only chooses one possible decomposition. Therefore, HYPERTREE always produces a join tree whose width is at most as large as CaT. Thus HYPERTREE generalizes CaT. For *triangle(3)* shown in Figure 12, HYPERTREE computes a hypertree with width 2 as shown in Figure 14, while CaT computes a join tree with width 3 when limiting cut size to 2, as shown in Figure 15. Note that here we denote s_1 as the hyperedge $\{P_1, P_2\}$, s_2 as the hyperedge $\{P_2, P_3\}$, s_3 as the hyperedge $\{P_1, P_3\}$, s_4 as the hyperedge $\{P_3, P_4\}$, s_5 as the hyperedge $\{P_4, P_5\}$, s_6 as the hyperedge $\{P_3, P_5\}$, s_7 as the hyperedge $\{P_5, P_6\}$, s_8 as the hyperedge $\{P_6, P_7\}$,

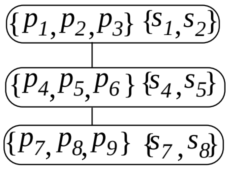

Fig. 14. Applying HYPERTREE on triangle(3).

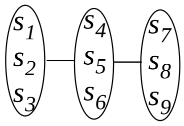

Fig. 15. Applying CaT on triangle(3).

and s_9 as the hyperedge $\{P_5, P_7\}$. Thus, HYPERTREE beats CaT. Therefore, HYPERTREE strongly generalizes CaT. □

Theorem 5. *HYPERTREE strongly generalizes TRAVERSE.*

Proof. HYPERTREE computes a join tree for a constraint hypergraph with the smallest hypertree width. TRAVERSE computes one possibility of an equivalent join tree for a constraint hypergraph. Therefore, for a same constraint hypergraph, HYPERTREE always produces a join tree whose width is smaller than or equal to the width of the join tree that TRAVERSE computes. HYPERTREE thus generalizes TRAVERSE. For H_{cg} shown in Figure 1, TRAVERSE computes a join tree with width 3, as shown in Figure 9, while HYPERTREE computes a join tree with width 2, as shown in Figure 11. Thus, HYPERTREE beats TRAVERSE. Therefore, HYPERTREE strongly generalizes TRAVERSE. □

Theorem 6. *HYPERTREE strongly generalizes HINGE$^+$.*

Proof. HYPERTREE computes a join tree for a constraint hypergraph with the smallest hypertree width. HINGE$^+$ computes one possible equivalent join tree of a constraint hypergraph. Therefore, for a same constraint hypergraph, HYPERTREE always produces a join tree whose width is smaller than or equal to the width of the join tree that HINGE$^+$ computes. Thus HYPERTREE generalizes HINGE$^+$. For H_{cg} shown IN Figure 1, HINGE$^+$ computes a join tree with width 4 when limiting cut size to 2, as shown in Figure 7, while HYPERTREE computes a hypertree tree with width 2, as shown in Figure 10. Thus, HYPERTREE beats HINGE$^+$. Therefore, HYPERTREE strongly generalizes HINGE$^+$. □

Theorem 7. *TRAVERSE and CUT are strongly incomparable.*

Proof. For H_{cg} shown in Figure 1, TRAVERSE computes a join tree with width 3, as shown in Figure 9, while CUT computes a join tree with width 4 when limiting cut size to 2, as shown in Figure 8. Thus, TRAVERSE beats CUT. For *book(4)* shown in Figure 13, a traverse decomposition from hyperedges $\{X_1, X, Y, Y_1\}$ has width 3, while a cut decomposition limiting cut size to 2 has width 2. Thus, CUT beats TRAVERSE. Therefore, TRAVERSE and CUT are strongly incomparable. □

Theorem 8. *TRAVERSE and HINGE$^+$ are strongly incomparable.*

Proof. For H_{cg} as shown in Figure 1, TRAVERSE computes a join tree with width 3, as shown in Figure 9, while HINGE$^+$ computes a join tree with width 5 when limiting cut size to 2, as shown in Figure 8. Thus, TRAVERSE beats HINGE$^+$. For *book(4)* shown in Figure 13, a traverse decomposition starting from hyperedges $\{X_1, X, Y, Y_1\}$ has width 3, while a hinge$^+$ decomposition limiting cut size to 2 has width 2. Thus, HINGE$^+$ beats TRAVERSE. Therefore, TRAVERSE and HINGE$^+$ are strongly incomparable. □

Figure 16 shows a constraint tractability hierarchy based on the above comparison results and Gottlob et al.'s conclusion [1]. This hierarchy is not yet complete.

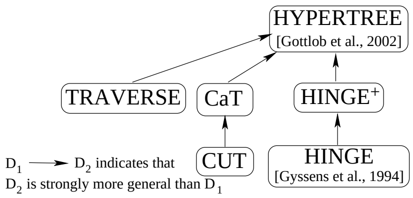

Fig. 16. Comparing our techniques to previous ones.

Table 1 summarizes the complexity of the decomposition techniques we discuss in this section.

Table 1. Complexity of decomposition methods.

Technique	Complexity
HYPERTREE	
Normal form: *opt-d-decomp* [7]	$O(\lvert\mathcal{S}\rvert^{2d}\lvert\mathcal{V}\rvert^2)$
Reduced normal form [10]	Best case: $O(\lvert\mathcal{S}\rvert^{d}\lvert\mathcal{V}\rvert + \lvert\mathcal{S}\rvert^2\lvert\mathcal{V}\rvert)$
HINGE	$O(\lvert\mathcal{V}\rvert\lvert\mathcal{S}\rvert^2)$
HINGE$^+$	$O(\lvert\mathcal{V}\rvert\lvert\mathcal{S}\rvert^{k+1})$
CUT	$O(\lvert\mathcal{V}\rvert\lvert\mathcal{S}\rvert^{k+1})$
TRAVERSE	$O(\lvert\mathcal{V}\rvert\lvert\mathcal{S}\rvert^2)$
CaT	$O(\lvert\mathcal{V}\rvert\lvert\mathcal{S}\rvert^{k+1})$
Solving the CSP after decomposition	$O(\lvert\mathcal{S}\rvert l^d d \log l)$

$|\mathcal{V}|$: 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 Conclusion

In this paper, we proposed new techniques to further improve the decomposition result of HINGE. We presented a different view of HINGE and generalize this method as HINGE^+. Then we introduced a variation of HINGE^+ called CUT. We also proposed a new decomposition method called TRAVERSE, which can be further combined with CUT as CaT. We showed that HINGE^+, CUT, TRAVERSE, and CaT can be performed in polynomial time. We compared HINGE, HINGE^+, CUT, TRAVERSE, CaT and HYPERTREE with each other. We enriched the constraint tractability hierarchy introduced by Gottlob et al. in [1] by adding the comparison results.

In the future, we plan to compare HINGE^+, CUT, TRAVERSE, and CaT with $\text{HINGE}^{\text{TCLUSTER}}$ and HINGE+BICOMP+HYPERTREE.

Acknowledgments: This work is supported by CAREER Award #0133568 from the National Science Foundation. Deb Derrick provided invaluable editorial help.

References

1. Gottlob, G., Leone, N., Scarcello, F.: A Comparison of Structural CSP Decomposition Methods. Artificial Intelligence **124** (2000) 243–282
2. Gyssens, M., Jeavons, P.G., Cohen, D.A.: Decomposing Constraint Satisfaction Problems Using Database Techniques. Artificial Intelligence **66** (1994) 57–89
3. Freuder, E.C., Hubbe, P.D.: A Disjunctive Decomposition Control Schema for Constraint Satisfaction. In Saraswat, V., Hentenryck, P.V., eds.: Principles and Practice of Constraint Programming. MIT Press, Cambridge, MA (1995) 319–335
4. Dechter, R., Pearl, J.: Tree Clustering for Constraint Networks. Artificial Intelligence **38** (1989) 353–366
5. Jeavons, P.G., Cohen, D.A., Gyssens, M.: A Structural Decomposition for Hypergraphs. Contemporary Mathematics **178** (1994) 161–177
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. Dechter, R.: Constraint Processing. Morgan Kaufmann (2003)
9. 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
10. 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