# Combining hypertree, bicomp, and hinge decomposition[1]

**Georg Gottlob** and **Martin Hutle**[2] and **Franz Wotawa**[3 4]

**Abstract.** Solving Constraint Satisfaction Problems (CSP) is in general NP-complete. If the structure of the CSP is a tree, then the computation can be done very effectively in a backtrack-free manner. There are several methods for converting CSPs in their tree-structured equivalent, e.g., hinge decomposition. More recently hypertree decomposition was developed and proved to subsume all other previously developed structure-based decomposition methods. In this paper we report recent results of a hypertree-decomposition implementation. We further have combined hypertree decomposition, biconnected component decomposition (bicomp), and hinge decomposition to improve running time and to make hypertree decomposition applicable on larger CSP instances. The formal requirements and the empirical results of the combined algorithms are reported.

## 1 Introduction

Solving Constraint Satisfaction Problems (CSP) is in general intractable but there are various subsets of CSPs that can be solved in polynomial time. Some of them can be identified by analyzing the structure of the CSP. If the CSP is tree-structured, then there exists a very efficient algorithm for solving the CSP. Therefore, much effort has been spent in compiling CSP to their tree-structured equivalent. These decomposition techniques include biconnected component [4] (BICOMP), hinge [8] (HINGE), and more recently hypertree decomposition [6, 5] (HYPERTREE). Gottlob and colleagues [7] compared different structural decomposition methods. They proved that hypertree decomposition is the most general method. Although, the complexity of hypertree decomposition is polynomial in the hypertree width, some problem instances can hardly be solved in reasonable time. Our empirical analysis showed that hypertree decomposition runs out of memory even for a CSP describing a 32-bit digital full adder (which comprises 160 constraints) on a SPARC Ultra 1 with 512 MB of main memory. The used implementation based on a slightly improved version of the `opt-k-decomp` algorithm given in [6]. There are other structural decomposition method we have not considered in our current work, e.g., tree clustering [3] and (hyper) cutset [2]. These methods should be considered in future extensions.

In this paper we describe the empirical results when applying `opt-k-decomp` to different problem instances, and show how the algorithm can be improved. The main idea of the combination is to first apply BICOMP on the original problem, then HINGE on the nodes of the resulting tree, and finally HYPERTREE. This saves time because the more time consuming decomposition methods are applied on smaller instances of the original problem. All of them compute the decomposition in polynomial time. From [7] we know that both BICOMP and HINGE are weaker than hypertree decomposition with respect to their width.

Our experimental results show that for some instances this combination of decomposition methods lead to a substantial improvement of running time. In order to combine the approaches we introduce an unified framework and describe the combination process in detail. Without restricting generality we assume that the hypergraphs of the CSPs are connected and reduced. Note that the connected subhypergraphs of such an unconnected hypergraph can be solved independently.

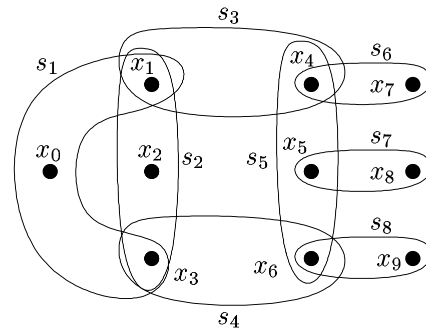


**Figure 1.** A hypergraph associated with a CSP (taken from [8])

This paper is organized as follows. In Section 2 and Section 3 we describe the biconnected component decomposition and the hinge decomposition respectively. In the next section we introduce the combination of BICOMP, HINGE, and HYPERTREE. After that we present empirical results of the decomposition techniques and their combination. Finally, we conclude the paper.

## 2 Biconnected component decomposition

A biconnected component of a graph $G = (V, E)$ is a maximal set of vertices $B \subseteq V$ such that its induced subgraph of $G$ is connected and remains connected after any one-vertex removal, i.e., $B$ has no separating vertex. In particular biconnected component can be a single vertex that is not element of another biconnected component. It is well known that all biconnected components of a graph $G$ can be computed in linear time. In [1] an algorithm for computing all biconnected components is given.

In order to make use of biconnected components for decomposing a CSP we first map the hypergraph $H$ that is given by the variables and the scopes of the constraints to its primal graph $G = (V, E)$. Any node $n \in V$ represents a variable of the CSP. Two nodes $n_1, n_2 \in V$ are connected if they are element of the same hypergraph edge. In the next step the biconnected components $C$ of the primal graph $G$ are

[1] This work was partially supported by Austrian Science Fund project N Z29-INF and DaimlerChrysler.
[2] Technische Universität Wien, Institut für Informationssysteme, Database and Artificial Intelligence Group, Favoritenstraße 9-11, A-1040 Vienna, Austria, email: {gottlob,hutle}@dbai.tuwien.ac.at
[3] Technische Universität Graz, Institute for Software Technology, Inffeldgasse 16b/II, A-8010 Graz, Austria, email: wotawa@ist.tu-graz.ac.at
[4] Authors are listed in alphabetical order.

computed. These biconnected components naturally induce a tree-structured decomposition $\langle T_B, \lambda_B, \chi_B \rangle$ where $\lambda_B$ maps nodes from $T_B$ to a set of associated constraints and $\chi_B$ maps nodes from $T_B$ to a set of associated variables. For convenience we further introduce the function $root$ returning the root of the decomposition, and the functions $parent$, $children$ returning the parent and the children of a given node, respectively.

A biconnected component decomposition of the given graph $G$ can be obtained as follows. Let $C$ be the biconnected components of $G$.

1. Let $S$ be the empty set.
2. Select an element $e$ from $C$ and remove it. Create a node $n$ with $\chi_B(n) = e$.
3. Add $n$ to $T_B$ and $S$, and let it be the root of the tree, i.e., $n = root$.
4. The tree is constructed in a breadth first manner. This is done by successively processing all nodes in $S$. Each node $n$ is processed as follows:
   (a) Remove $n$ from $S$.
   (b) For each $e' \in C$ with $e' \cap \chi_B(n) \neq \emptyset$ do:
      i. Create a new node $n'$ and add it to $T_B$ and $S$. This node will be processed after all other nodes in $S$ are processed.
      ii. Update the data structure. Let $children(n') = \emptyset$, $parent(n') = n$, $\chi_B(n') = e'$, and add $n'$ to $children(n)$.
5. For each node $n \in T_B$ update the $\lambda_B$ value. This value is given by $\lambda_B(n) = \{(x_1, \ldots, x_n) | (x_1, \ldots, x_n) \in H \wedge \{x_1, \ldots, x_n\} \subseteq \chi_B(n)\}$.
6. Return $\langle T_B, \lambda_B, \chi_B \rangle$.

Figure 2 shows he result of BICOMP when applied on the hypergraph which is given in Figure 1. We see that the only variable that is shared between the vertex $\{s_1, s_2, s_3, s_4, s_5\}$ and vertex $\{s_6\}$ is $x_4$. The same holds for other connected vertices as specified by the following propositions.

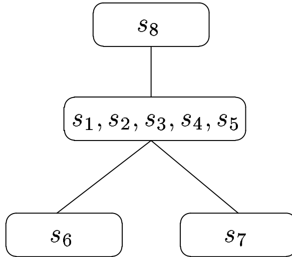


**Figure 2.** The resulting tree after applying BICOMP.

**Proposition 2.1** *Let $\langle T_B, \lambda_B, \chi_B \rangle$ be a biconnected component decomposition and $n, n_1, n_2 \in T_B$ be nodes where $n$ is not the root node and $n_1 \neq n_2$.*

- *$n$ does exactly share one variable with its parent.*

$$\chi_B(n) \cap \chi_B(parent(n)) = \{x\}$$

- *No two different nodes share the same constraints.*

$$\lambda_B(n_1) \cap \lambda_B(n_2) = \emptyset$$

From the definition of biconnected components the following proposition follows immediately.

**Proposition 2.2** *Let $\langle T_B, \lambda_B, \chi_B \rangle$ be a biconnected component decomposition and let $n \in T_B$ be a node, then $\bigcup_{(x_1,\ldots,x_m) \in \lambda_B(n)} \{x_1, \ldots, x_m\} = \chi_B(n)$.*

There are no elements in $\chi_B$ that are not covered by a constraint, i.e., a hyper-edge, in $\lambda_B$. Moreover, the connectedness condition that holds for every node of a hypertree decomposition also holds for nodes of a biconnected component decomposition. The connectedness condition says that if a variable $x$ is element of $\chi_B(n)$ and $\chi_B(n')$, then $x$ must be element of $\chi_B(n'')$ where $n''$ lies on the path between $n$ and $n'$. This condition must be true because of following observation. Assume variable $x$ occurs in $\chi_B(n_i)$ of one or more nodes $n_1, \ldots, n_k$. If $k = 1$, the variable $x$ cannot be element of $\chi_B$ of another node $n' \neq n$ because of the definition of biconnected components. Otherwise, there must be a direct connection between the nodes $n_1, \ldots, n_k$. Hence, the condition is valid.

**Lemma 2.3** *The connectedness condition holds for the biconnected component decomposition.*

This lemma is important to ensure that a hypertree decomposition can be combined with biconnected component decomposition.

## 3 Hinge decomposition

Gyssen et al. [8] introduced the concept of hinge decomposition for decomposing CSPs. HINGE can be seen as an extension of BICOMP, since HINGE makes use of separating edges instead of separating vertices. Hinges are defined as follows:

**Definition 3.1 (Hinge)** *Let $(V, E)$ be a reduced and connected hypergraph, and let $H$ be either $E$ or a proper subset of $E$ containing at least two edges. Let $H_1, \ldots, H_m$ be the connected components of $E - H$ with respect to $H$. The $H$ is called a hinge if, for $i = 1, \ldots, m$, there exists an edge $h_i$ in $H$ so that $(\bigcup H_i) \cap (\bigcup H) \subseteq h_i$. The edge $h_i$ is called a separating edge for $H_i$.*

This definition makes use of the notation of connected components with respect to a set $H \subseteq E$. A set $F \subseteq E - H$ is called connected with respect to $H$ if, for any two edges $e, f \in F$, there exists a sequence $e_1, \ldots, e_n$ of edges in $F$ such that (i) $e_1 = e$; (ii) for $i = 1, \ldots, n-1$, $e_i \cap e_{i+1}$ is not contained in $\bigcup H$; and (iii) $e_n = f$. The maximal connected subsets of $E - H$ with respect to $H$ are called the connected components of $E - H$ with respect to $H$.

A separating edge partitions the hypergraph into (one or more) sub-hypergraphs which are connected only by this edge. Therefore, when we compute all minimal hinges, i.e., hinges that do not contain any other hinge, we can build up a tree with the hinges as vertices that are connected by their separating edge.

**Definition 3.2 (Hinge-tree)** *Let $(V, E)$ be any hypergraph. A hinge-tree of $(V, E)$ is a tree $(N, A)$ with nodes $N$ and labeled arcs $A$, so that:*

1. *The tree nodes are minimal hinges of $(V, E)$.*
2. *Each edge in $E$ is contained in at least one tree node.*
3. *Two adjacent tree nodes share precisely one edge of $E$ which is also the label of their connecting tree arc; moreover, their shared vertices are precisely the members of this edge.*
4. *The vertices of $V$ shared by two tree nodes are entirely contained within each tree node on their connecting path.*

An algorithm for computing hinge-trees can be found in [8]. Computing a hinge-tree is of order $O(|V| \cdot |E|^2)$ when representing the

edges as bit arrays. When storing the vertices of an edge as a sequential collection (e.g., a linked list, array) $O(m \cdot |E|^2)$ can be achieved, where $m$ is the maximal number of vertices in an edge.

The hinge-tree of the example CSP from Figure 1 is depicted in Figure 3. Hinge-trees can also be represented as hypertrees $\langle T_H, \lambda_H, \chi_H \rangle$, with:

- $T_H$ are all nodes of the hinge-tree.
- $\lambda_H$ of a node is equal to the edges (e.g., constraints) stored in the node.
- $\chi_H$ of a node is equal to the variables of the node's edges.

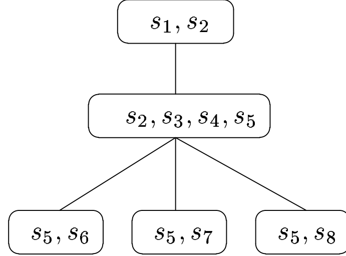


**Figure 3.** A resulting tree after applying `HINGE`.

From the definition of hinge-trees follows immediately that hinge-trees and their equivalent hypertrees fulfill the connectedness condition.

## 4 Combining biconnected component, hinge, and hypertree decomposition

In this section we introduce how the considered decomposition techniques can be combined. The basic idea is to first use the result of one decomposition technique and then apply the other technique on the subproblems that are induced by the $\chi$ sets of the nodes. What we get is a incremental refinement of the hypertree. Due to the hierarchy of decomposition methods given in [7] it make only sense to apply `BICOMP` on the original hypergraph. Take the result and apply `HINGE` (to all biconnected components), and finally apply `HYPERTREE` (to all hinges).

The following algorithm takes a hypertree that is computed by one decomposition method and applies the second decomposition method on the resulting nodes. In order to get a correct decomposition the algorithm has to take care of the intersection variables between the $\chi$ sets of connected nodes.

**Algorithm** `combine`$(D_1, D_2, H)$
**Input:** 2 decomposition methods $D_1, D_2$, where $D_1$ is weaker than $D_2$ with respect to their width; $H$ the hypergraph of a CSP.
**Output:** A hypertree representing the decomposed CSP

1. Take $H$ and compute the decomposition $\langle T_1, \lambda_1, \chi_1 \rangle$ with method $D_1$.
2. Return `travTree` $(< T_1, \lambda_1, \chi_1 >, D_2)$

**Algorithm** `travTree`$(< T_1, \lambda_1, \chi_1 >, D_2)$
**Input:** A hypertree $< T_1, \lambda_1, \chi_1 >$; a decomposition method $D_2$.
**Output:** A hypertree representing the decomposed CSP

1. Let $\langle T, \lambda, \chi \rangle$ be an empty decomposition.
2. Starting from the root of $T_1$ traverse the tree in a breadth first manner and process each node. A node $n$ is processed as follows:
   (a) Compute a decomposition $\langle T_2, \lambda_2, \chi_2 \rangle$ for the CSP that is induced by $\lambda_1(n)$ using method $D_2$. If $n$ is not the root node of $T_1$, then the $\chi_2$ value of the root node of $T_2$ must contain the elements $M$, where $M = \chi_1(n) \cap \chi_1(parent(n))$.
   (b) Add all vertices and arcs of $T_2$, the entries of $\chi_2$ and $\lambda_2$ to $\langle T, \lambda, \chi \rangle$.
   (c) If $n$ is not the root node, make an arc in $T$ from $root(T_2)$ to a node $m$ of the $D_2$ decomposition of $parent(n)$ (which is an element of $T$) where $M \subseteq \chi_2(m)$. If $root(T_2)$ and $m$ have the same $\chi$ value, then $root(T_2)$ can be removed from the resulting decomposition and all arcs going to $root(T_2)$ are redirected to $m$.
3. Return the resulting decomposition $\langle T, \lambda, \chi \rangle$.

Figure 4 shows the result when applying one decomposition $D_2$ to the result of another decomposition $D_1$. The root node of the converted subtree of node $n$ must have the variables $M$ in its $\chi$ set, i.e., $M \subseteq \chi(parent(n))$. One of the nodes that have the variables $M'$ in their $\chi$ set must be connected with the root nodes of the converted trees of $n$'s children.

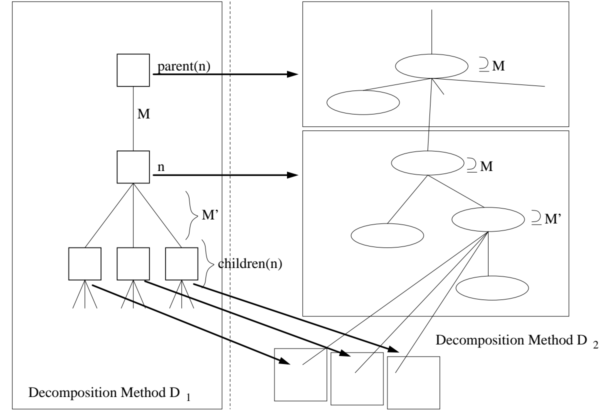


**Figure 4.** The idea behind `combine` and `travTree`

The `combine` algorithm can be extended to combine an arbitrary number of decomposition methods. Assume $D_1, \ldots, D_n$ where $D_i$ is weaker than $D_{i+1}$ with respect to their width. The general combination algorithm `combine+` is given by:

$$\texttt{combine+}(D_1, \ldots, D_n, H) = \\ = \begin{cases} D_1(H) & \text{if n=1} \\ \texttt{travTree}(D_n, \texttt{combine+}(D_1, \ldots, D_{n-1}, H)) & \text{otherwise} \end{cases}$$

In the empirical results section the combination of all three decomposition method is called `HYBRID`. `HYBRID` is defined as `combine+(HYPERTREE,BICOMP,HINGE,`$H$`)` where $H$ is the original hypergraph of the CSP.

In the following we argue about the correctness of `combine`.

**Definition 4.1 (OEIC property)** *A decomposition method $D$ for a CSP is said to have the One Edge Interface Covering (OEIC) property iff the intersection of the variables of an arbitrary vertex $n$ and*

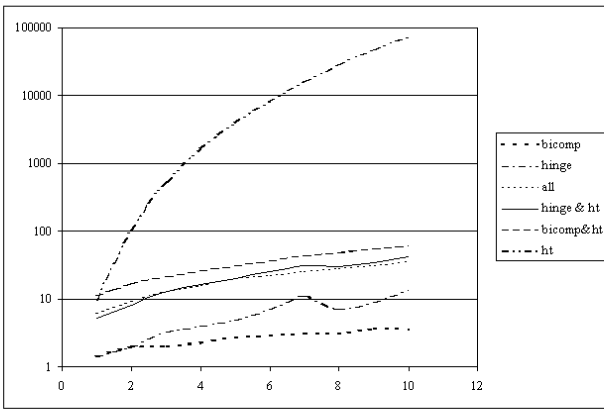


**Figure 5.** The adder example decomposed by different decomposition strategies

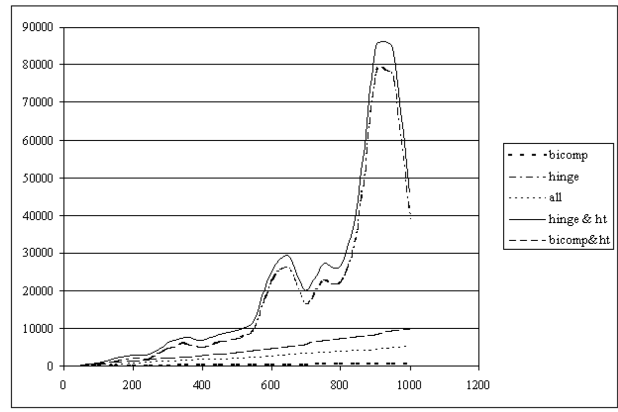


**Figure 6.** The adder example for larger problem sizes

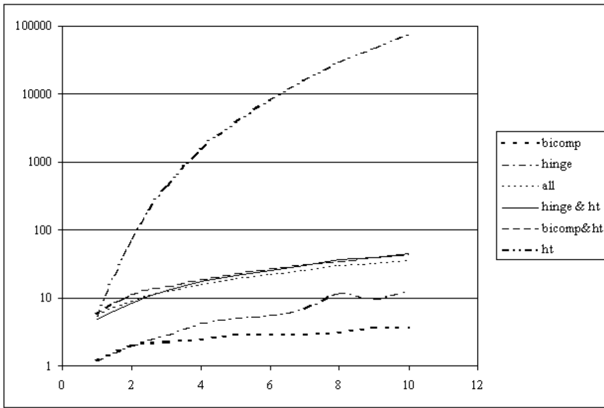


**Figure 7.** The switch&bulb example decomposed by different decomposition methods

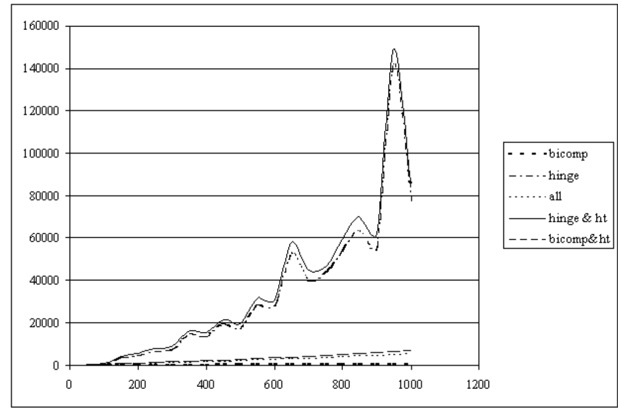


**Figure 8.** The switch&bulb example for larger problem sizes

*its parent are completely covered by one edge in* $\lambda(n)$ *and one edge in* $\lambda(parent(n))$.

The OEIC property of the weaker decomposition method $D_1$ of the combination algorithm is important to ensure that an arc can always be added to the resulting tree during computation in step 2c of the `travTree` algorithm. However, this property is not sufficient. It must also be guaranteed that it is always possible to compute a decomposition of the induced sub-problem of a vertex $n$ (that is given by its edges) where the root captures all intersecting variables of $n$ and $n$'s parent.

**Theorem 4.1** *Let* $D_1$, $D_2$ *be two decomposition methods. A CSP can be correctly decomposed using* `combine` *iff:*

- $D_1$ *has the OEIC property.*
- $D_2$ *returns a tree whose root contains a given hyperedge of the hypergraph to be decomposed.*

It is obvious that both conditions of Theorem 4.1 hold for `BICOMP` and `HINGE`. But the second condition does not hold in general for minimal width hypertree decomposition, or for hypertree decompositions of a given width. The requirement that the variables of a certain edge must appear in the root may, in general, increase the width of a decomposition. However, in the special case where for each node $n$, $\chi(n)$ always contains all variables of $n$, the second condition is satisfied. If there is no solution of `HYPERTREE` of a given width where the second condition satisfied, then there must be a decomposition of a larger width where the condition is true. In the worst case the hypertree width of the decomposition is equal to the number of edges of the problem. However, in all examples we have used to test `combine` it was always possible to find a decomposition that captures the given variable set.

## 5 Empirical results

We implemented our approach in Cincom Smalltalk VisualWorks 5i.4 to gain empirical results for the decomposition methods and their combination. For this purpose we used three scalable CSP examples, i.e., a digital full adder, an electronic bridge circuit, and a switch&bulb example. The adder is a model of a full adder where the constraints represent the logic gates. By connecting the outgoing carry of such an adder with another instance of the adder we get a sequence of full adders of arbitrary length. The bridge circuit is formed by resistors. There the sequence is achieved also by connecting their inputs and outputs. The switch&bulb example is another electrical circuit where the single elements of the sequence are loosely coupled by a bulb that drives a photo-resistor. These examples have the major advantage that they are scalable. Therefore, they allow us to test our approach on problems of different sizes.

All measures were taken on a Windows 2000 Workstation with 1.5GB RAM and an AMD Athlon 1.2 GHZ CPU. Each running time was measured five times and averaged to obtain a single value. The running times are given in milli-seconds. In order to avoid garbage collecting of the Smalltalk engine during the tests, we invoke the system's garbage collector each time before starting a single measurement run.

**The full adder example:** For the first tests we measured how different decomposition methods perform on the adder example. Let's consider a sequence of $n$ instances, i.e. an $n$-bit adder. BICOMP and HINGE can decompose the problem very well so that for increasing $n$ the bicomp- and the hinge-width remains constant. Therefore, we have a very good performance for our HYBRID decomposition method. The empirical results obtained for 1- to 10-bit adders are given in Figure 5. Note that the y-axis shows the execution time in milliseconds on a logarithmic scale. The x-axis shows the number of bits of the adder. Since each adder comprises of five constraints this value has to be multiplied by five to yield the number of constraints.

BICOMP and HINGE perform much better than HYPERTREE with increasing number of bits. The hypertree decomposition can - in conformity with the theory - easily be identified with a complexity lower than exponential (which would be a straight line in the logarithmic scale). The hybrid decomposition methods require a little bit more time than BICOMP and HINGE because they had to perform additional computations. However, the additional required running time is the same for all hinges respective biconnected components. Therefore, we only get a small linear overhead. The fluctuations of the curves for small time measures are due to inaccuracies of these small intervals.

To get rid of this and better analyze the different hybrid methods we took HYPERTREE out of the race and increased the number of full adders by 50 for each step (see Figure 6). Now the curves part into two groups: those that use BICOMP (with linear running time) and those that use only HINGE (with cubic running time). Note that the fluctuations of HINGE and HINGE+HYPERTREE are not due to incorrect measurements, indeterministic influences, or system loads. All values are averaged over 5 measures and both methods have the peaks at the same points. In addition we measured first all methods for one instance size and then proceeded with the next size. Since the other methods had not this peaks we expect that implementation specific parameters generate this peaks. In particular the hash and grow functions of the Smalltalk set implementation are good candidates as the source of this phenomenon.

**The Switch and Bulb Example:** Like the adder example, this scalable problem can be decomposed by BICOMP and HINGE. Therefore, the results are very similar to those of the adder. Figure 7 shows the results for all methods and sequences of 1 to 10 switch and bulb circuits. Figure 8 omits HYPERTREE and yields the running time results for larger instance sizes.

**The Bridge Example:** The Bridge example can be separated only into 7 hinges, independent of the size. 6 of those hinges are just small subproblems. Therefore, there is one large hinge that contains almost the whole hypergraph. So HYBRID is just a little bit faster than HYPERTREE. The measurements underpin this observation. Only HINGE and BICOMP are significant faster than the hypertree decomposition but they lead to decompositions with a width that depend on the CSP size. I.e., if the instances become larger the width increases as well.

Finally, we compare the different examples decomposed with a full hybrid decomposition (BICOMP, HINGE, HYPERTREE). What we see is that all times BICOMP and HINGE allow to decompose the original problem in smaller sub-problems, HYBRID shows a very good running time performance. If BICOMP and HINGE do not lead to sub-problems of a small width, there is almost no performance gain. However, since both decomposition techniques can be performed in a very short time compared to HYPERTREE it is always a good idea to make use of them. Although in cases where HYPRID does not lead to a substantially better running time performance, it performs equally well than HYPERTREE alone (see the results for our bridge circuit example).

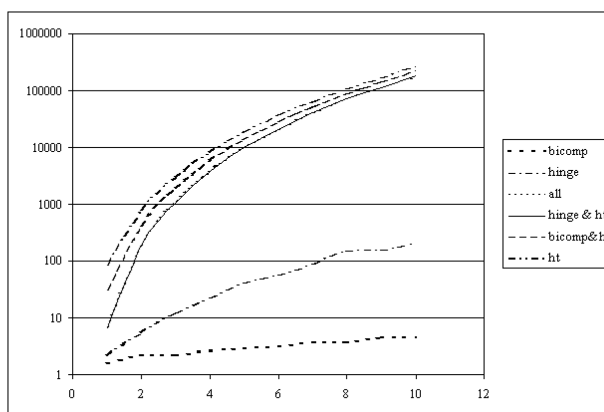


**Figure 9.** The performance of different decomposition methods on the bridge example

## 6 Conclusion

In this paper we introduce a framework for combining hypertree decomposition, hinge decomposition, and biconnected components decomposition. Moreover, we presented first empirical results that show that the combined HYPRID method improves running time and allows the application of hypertree decomposition on larger problem instances. Although, performance gain of HYPRID depends on the underlying structure of the CSP, the overhead costs are small. Even for CSPs where HINGE or BICOMP cannot be applied well, the running time of the HYPRID method is almost equal to the running time of HYPERTREE alone. Since the proposed combination does not guarantee to work on all CSP instances, future research should deal with improving opt-k-decomp with respect to its space and time requirements.

## REFERENCES


[1] A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley Publishing Company, 1974.

[2] Rina Dechter, 'Constraint networks', in *Encyclopedia of Artificial Intelligence*, 276–285, Wiley and Sons, (1992).

[3] Rina Dechter and Judea Pearl, 'Tree clustering for constraint networks', *Artificial Intelligence*, **38**, 353–366, (1989).

[4] Eugene C. Freuder, 'A Sufficient Condition for Backtrack-Bounded Search', *Artificial Intelligence*, **32**(4), 755–761, (1985).

[5] 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)*, pp. 21–32, Philadelphia, PA, (1999).

[6] 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).

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

[8] Marc Gyssens, Peter G. Jeavons, and David A. Cohen, 'Decomposing constraint satisfaction problems using database techniques', *Artificial Intelligence*, **66**, 57–89, (1994).