A Backtracking-Based Algorithm for Hypertree Decomposition

GEORG GOTTLOB
University of Oxford
and
MARKO SAMER
University of Durham

Hypertree decompositions of hypergraphs are a generalization of tree decompositions of graphs. The corresponding hypertree-width is a measure for the acyclicity and therefore an indicator for the tractability of the associated computation problem. Several NP-hard decision and computation problems are known to be tractable on instances whose structure is represented by hypergraphs of bounded hypertree-width. Roughly speaking, the smaller the hypertree-width, the faster the computation problem can be solved. In this paper, we present the new backtracking-based algorithm `det-`k`-decomp` for computing hypertree decompositions of small width. Our benchmark evaluations have shown that `det-`k`-decomp` significantly outperforms `opt-`k`-decomp`, the only exact hypertree decomposition algorithm so far. Even compared to the best heuristic algorithm, we obtained competitive results as long as the hypergraphs are sufficiently simple.

Categories and Subject Descriptors: F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems—*Computations on discrete structures*; G.2.1 [**Discrete Mathematics**]: Combinatorics—*Combinatorial algorithms*; G.2.2 [**Discrete Mathematics**]: Graph Theory—*Hypergraphs, Trees*; I.2.8 [**Artificial Intelligence**]: Problem Solving, Control Methods, and Search—*Backtracking, heuristic methods*

General Terms: Algorithms, Design, Experimentation

Additional Key Words and Phrases: Constraint satisfaction, hypertree decomposition

ACM Reference Format:

Gottlob, G. and Samer, M. 2008. A backtracking-based algorithm for hypertree decomposition. ACM J. Exp. Algor. 13, Article 1.1 (September 2008), 19 pages DOI 10.1145/1412228.1412229 http://doi.acm.org 10.1145/1412228.1412229

Research supported by the Austrian Science Fund (FWF), project P17222-N04.

Authors' addresses: Georg Gottlob, Oxford University Computing Laboratory, Wolfson Building, Parks Road, Oxford OX1 3QD, England, UK; email: georg.gottlob@comlab.ox.ac.uk. Marko Samer, Department of Computer Science, Durham University, Science Labs, South Road, Durham DH1 3LE, England, UK; email: marko.samer@durham.ac.uk.

1. INTRODUCTION

Since many important problems in computer science are, in general, intractable, it is a competitive task to identify tractable subclasses of such problems. One approach to do this is to restrict the structure of a problem represented as a graph or a hypergraph. For example, the structure of instances of the constraint satisfaction problem CSP and the equivalent Boolean conjunctive query problem BCQ can be naturally represented by a hypergraph. Hypergraphs are a generalization of graphs where each edge connects a set of vertices. Gottlob et al. [2002] have shown that, in analogy to treewidth of graphs, the hypertree-width of hypergraphs is an appropriate measure for the acyclicity and therefore an indicator for the tractability of the corresponding computation problems. The *hypertree-width* of a hypergraph is defined as the minimum width over all *hypertree decompositions* of the hypergraph (see Section 2 for a formal definition). Roughly speaking, the smaller the width of a hypertree decomposition, the faster the corresponding problem can be solved. However, deciding whether there exists a hypertree decomposition of width at most k is NP-complete, in general.

Gottlob et al. [2002] have shown that for fixed k, the problem of deciding whether there exists a hypertree decomposition of width, at most, k is in LogCFL, i.e., it can be solved in polynomial time and is highly parallelizable. Moreover, they presented the alternating algorithm k-`decomp`, which constructs a hypertree decomposition of minimal width less than or equal to k (if such a decomposition exists). Another algorithm for computing hypertree decompositions of minimal width less than or equal to k is `opt-`k`-decomp` [Gottlob et al. 1999; Leone et al. 2002; Scarcello et al. 2007]. So far, `opt-`k`-decomp` has been the only implementable exact polynomial-time algorithm for constructing k-bounded hypertree decompositions. However, `opt-`k`-decomp` has an important disadvantage: although polynomial, it needs a huge amount of memory and time even for small hypergraphs. This basic problem remains even after improvements like redundancy elimination, as investigated by Harvey and Ghose [2003]. Thus, recent research focuses on heuristic approaches for constructing hypertree decompositions of small, but not necessarily minimal, width. One of the practically most successful algorithms of this kind has been developed by McMahan [2004], who combined well-known tree decomposition and set cover heuristics in order to construct hypertree decompositions. This approach, however, has the disadvantage that the computation cannot be focused on hypertrees of width smaller than some upper bound k, i.e., the algorithm returns a hypertree decomposition whose width may be much larger than the minimal one, although there is often time left to improve this result.

In this paper, we consider a combination of exact and heuristic approaches in the sense that we restrict the search space by a fixed upper bound k and apply heuristics to accelerate the search for a hypertree decomposition of width, at most, k (but not necessarily the minimal one). Such an approach has the following advantages: the search space and the width of the resulting hypertree decomposition can be bounded by k. In particular, this means that the algorithm is able to find a hypertree decomposition of minimal width by setting k

small enough. On the other hand, if k is not minimal, the algorithm may construct a hypertree decomposition of nonminimal width, but within a reasonable amount of time since the solution space becomes larger. In principle, such an approach allows us to control the tradeoff between the resulting hypertree-width and the required computation time. Thus, it combines the advantages of exact and heuristic algorithms while it minimizes their disadvantages. Evidently, we could have implement this approach with a plain backtracking-based search procedure without heuristics; however, experimental results have shown that the computation times become very large without the use of heuristics.

We implemented our algorithm `det-`k`-decomp` based on these insights. Our experimental evaluations have shown that it performs much better than `opt-`k`-decomp`. In particular, it needs only a small amount of memory and is much faster than `opt-`k`-decomp`. Moreover, it is often the case that `det-`k`-decomp` finds hypertree decompositions of width smaller than those obtained by McMahan's heuristic algorithm. Finally, there is a further advantage of our algorithm: because of its top-down nature of decomposing a hypergraph into subhypergraphs and decomposing the subhypergraphs into subsubhypergraphs, etc., it can, in principle, be implemented for parallel execution in order to increase its performance even further.

This paper is organized as follows: In Section 2, we give the basic definitions used in this paper. Afterward, in Section 3, we describe the alternating algorithm k`-decomp`. Then, in Section 4, we introduce our new algorithm `det-`k`-decomp` and prove its correctness, polynomial runtime, and space requirement. In Section 5, we present our experimental results. Finally, we conclude in Section 6.

2. PRELIMINARIES

A *hypergraph* H is a tuple (V, E), where V is a set of vertices and $E \subseteq 2^V \setminus \{\emptyset\}$ is a set of hyperedges; w.l.o.g., we assume that $V \setminus \bigcup E = \emptyset$. We put $\mathit{vertices}(H) = V$ and $\mathit{edges}(H) = E$. A *hypertree for a hypergraph* H is a triple (T, χ, λ), where $T = (V, E)$ is a tree and $\chi : V \to 2^{\mathit{vertices}(H)}$ and $\lambda : V \to 2^{\mathit{edges}(H)}$ are labeling functions. We put $\mathit{vertices}(T) = V$ and refer to the vertices of T as "nodes" to avoid confusion with the vertices of H. For a subtree $T' = (V', E')$ of T, we put $\chi(T') = \bigcup_{p \in V'} \chi(p)$. We denote the root of T by $\mathit{root}(T)$, and for every $p \in \mathit{vertices}(T)$ we denote the subtree of T rooted at p by T_p.

A *hypertree decomposition* of a hypergraph H is a hypertree (T, χ, λ) for H satisfying the following four conditions [Gottlob et al. 2002]:[1]

1. $\forall e \in \mathit{edges}(H)\ \exists p \in \mathit{vertices}(T) : e \subseteq \chi(p)$,
2. $\forall v \in \mathit{vertices}(H)$: the set $\{p \in \mathit{vertices}(T) \mid v \in \chi(p)\}$ induces a (connected) subtree of T,

[1]A hypertree satisfying only the first three conditions of a hypertree decomposition is called a *generalized hypertree decomposition*. Recently, it was shown that even for fixed k deciding whether a hypergraph has generalized hypertree-width, at most, k is NP-hard [Gottlob et al. 2007].

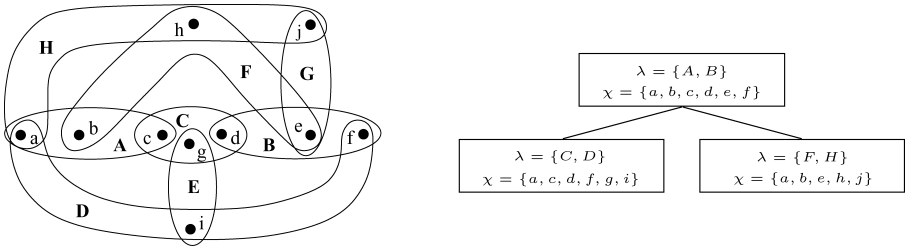


Fig. 1. Example of a hypergraph and its hypertree decomposition of width 2.

3. $\forall p \in \mathit{vertices}(T) : \chi(p) \subseteq \bigcup \lambda(p)$, and
4. $\forall p \in \mathit{vertices}(T) : \bigcup \lambda(p) \cap \chi(T_p) \subseteq \chi(p)$.

The *width of a hypertree decomposition* (T, χ, λ) is given by $\max_{p \in \mathit{vertices}(T)} |\lambda(p)|$, and the *hypertree-width of a hypergraph* is the minimal width over all its hypertree decompositions. Figure 1 shows an example of a hypergraph (on the left) and its hypertree decomposition of width 2 (on the right). Note that the above definition does not require that for each hyperedge e there exists a node $p \in \mathit{vertices}(T)$ such that $e \in \lambda(p)$ and $e \subseteq \chi(p)$. A hypertree decomposition satisfying this additional condition is called *complete*. Every hypertree decomposition of width k can be transformed in logarithmic space into a complete hypertree decomposition of width k [Gottlob et al. 2002].

Let $H = (V, E)$ be a hypergraph. A path between two vertices $x, y \in V$ is a sequence of vertices $x = v_1, v_2, \ldots, v_{k-1}, v_k = y$ such that for all pairs v_i, v_{i+1} there exists $e \in E$ with $v_i, v_{i+1} \in e$. Now, let $W \subseteq V$. A set $V' \subseteq V$ is $[W]$-connected if for all $x, y \in V'$ there exists a path between x and y not going through vertices in W. A $[W]$-*component* is a maximal $[W]$-connected nonempty set $V' \subseteq V \setminus W$ of vertices. We call the set W in this context a *separator*. Note that both components and separators can also be represented by the hyperedges covering the corresponding vertices, which we will do in this paper.

3. THE ALGORITHM *K*-DECOMP

In this section, we describe the alternating algorithm `k-decomp` introduced by Gottlob et al. [2002]. This algorithm checks nondeterministically whether there exists a hypertree decomposition of a hypergraph of width, at most, k. By using this algorithm, Gottlob et al. proved that the problem of deciding whether a hypergraph has k-bounded hypertree-width is in LogCFL, a very low polynomial-time complexity class, which is contained in NC_2 and which thus consists of highly parallelizable problems. The algorithm works in a top-down manner by guessing in each step the λ-labels and checking two conditions. The χ-labels and subhypergraphs can then be computed deterministically. A game-theoretic characterization of this approach can be found in [Gottlob et al. 2003].

The algorithm `k-decomp` shown in Algorithms 1 and 2 is a notational variant of the algorithm presented by Gottlob et al. [2002]. A proof of equivalence of the two variants is given in the appendix. The main procedure `k-decomp` in Algorithm 1 expects a hypergraph `HGraph` consisting of a set of vertices `vertices(HGraph)` and a set of hyperedges `edges(HGraph)` as parameter.

Algorithm 1 *k-decomp*(*HGraph*).

```
1 HTree := k-decomposable (edges (HGraph), ∅);
2 return HTree;
```

Algorithm 2 *k-decomposable*(*Edges*, *OldSep*).

```
1 guess Separator ⊆ edges (HGraph) such that |Separator| ≤ k;
2 check whether the following two conditions hold:
3     ⋃Edges ∩ ⋃OldSep ⊆ ⋃Separator;
4     Separator ∩ Edges ≠ ∅;
5 if one of these checks fails then return NULL;
6 Components := separate (Edges, Separator);
7 Subtrees := ∅;
8 for each Comp ∈ Components do
9     HTree := k-decomposable (Comp, Separator);
10    if HTree = NULL then
11        return NULL;
12    else
13        Subtrees := Subtrees ∪ {HTree};
14    endif
15 endfor
16 Chi := (⋃Edges ∩ ⋃OldSep) ∪ ⋃ (Separator ∩ Edges);
17 HTree := getHTNode (Separator, Chi, Subtrees);
18 return HTree;
```

It calls the recursive procedure `k-decomposable` in Algorithm 2, which returns a hypertree decomposition of width, at most, k if it exists and `NULL` otherwise. The parameters of `k-decomposable` are a set `OldSep` of hyperedges, which were chosen as separator in the previous run and a set `Edges` of hyperedges, which represents the current subhypergraph that has to be decomposed (i.e., `Edges` represents a component w.r.t. `OldSep`). In line 1, a set `Separator` of size, at most, k is guessed. In lines 2–4, it is then checked whether `Separator` is indeed a separator, i.e., whether it decomposes the current subhypergraph into several (at least one) smaller components and whether the conditions of a hypertree decomposition are satisfied. In line 6, the procedure `separate` computes these components and stores them in `Components`. Afterward, in lines 7–15, the hypertree decompositions of the components are recursively computed and stored in `Subtrees`. Line 16 computes the χ-labels of the current hypertree node in such a way that the conditions of a hypertree decomposition are satisfied. Finally, the procedure `getHTNode` in line 17 constructs the current hypertree node consisting of the λ-labels in `Separator`, the χ-labels in `Chi`, and the subtrees in `Subtrees`.

For example, when applying `k-decomp` to the hypergraph in Figure 1, `k-decomposable` is called with parameters `Edges` $= \{A, B, C, D, E, F, G, H\}$ and `OldSep` $= \emptyset$. In the first run, we guess `Separator` $= \{A, B\}$, which results in `Components` $= \{\{C, D, E\}, \{F, G, H\}\}$. Consequently, there are two recursive calls: the first one uses parameters `Edges` $= \{C, D, E\}$ and `OldSep` $= \{A, B\}$, and the second one uses `Edges` $= \{F, G, H\}$ and `OldSep` $= \{A, B\}$. In the

Algorithm 3 *det-k-decomp* (*HGraph*)

```
1 FailSeps := ∅;
2 SuccSeps := ∅;
3 HTree := decompCov(edges(HGraph), ∅);
4 if HTree ≠ NULL then
5     HTree := expand(HTree);
6 endif
7 return HTree;
```

following recursion level, we guess Separator $= \{C, D\}$ and Separator $= \{F, H\}$, respectively. It is easy to see that no further recursive calls are necessary, since both subhypergraphs are then completely decomposed. The resulting hypertree decomposition is shown in Figure 1.

As it was shown by Gottlob et al. [2002], k-decomp can be implemented on a logspace alternating Turing machine having polynomially bounded tree-size. Hence, deciding whether a hypergraph has k-bounded hypertree-width is in LOGCFL and thus polynomially solvable. However, since k-decomp is a nondeterministic algorithm, this is only of theoretical interest and cannot be implemented. The only deterministic algorithm with polynomial runtime for computing k-bounded hypertree decompositions published so far is opt-k-decomp [Gottlob et al. 1999; Leone et al. 2002; Scarcello et al. 2007]. This algorithm, however, has a basic drawback: because of its bottom-up approach, it needs a huge amount of memory even for small instances. Thus, it is practically only applicable to hypergraphs of very small size and simple structure. Our aim in the following section therefore, is to transform k-decomp into a deterministic algorithm with polynomial runtime, which performs better than opt-k-decomp. We call our resulting algorithm det-k-decomp.

4. THE ALGORITHM DET-*K*-DECOMP

In this section, we present our new algorithm det-k-decomp, which we obtain from k-decomp by replacing the *guess and check*-part in lines 1–4 of Algorithm 2 by a backtracking-based search procedure. In order to preserve the polynomial runtime, however, we have to store already visited separators; otherwise, the same subtree could be constructed multiple times, which would lead to an exponential runtime. Algorithm 3 represents the main procedure which calls the recursive procedures decompCov, decompAdd, and decompSub described in Algorithms 4–6. Roughly speaking, decompCov computes the λ-labels by locally covering a fixed subset of the χ-labels, decompAdd adds a hyperedge to the λ-labels if necessary to enforce the subcomponents to shrink monotonically, and decompSub decomposes the subcomponents recursively. We divided the whole algorithm into these subprocedures for better readability. Their dependencies are shown in Figure 2, where an arrow from A to B means that A calls B. Moreover, we use three auxiliary procedures:

—separate is the same as for k-decomp, described in Section 3, and computes the components of the current subhypergraph w.r.t. the chosen separator.

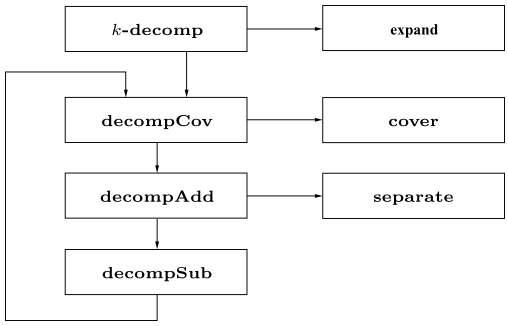


Fig. 2. Outline of the procedure calls of `det-k-decomp`.

Algorithm 4 *decompCov* (*Edges*, *Conn*)

```
1 if |Edges| ≤ k then
2     HTree := getHTNode (Edges, ⋃Edges, ∅);
3     return HTree;
4 endif
5 BoundEdges := {e ∈ edges (HGraph) | e ∩ Conn ≠ ∅};
6 for each CovSep ∈ cover (Conn, BoundEdges) do
7     HTree := decompAdd (Edges, Conn, CovSep);
8     if HTree ≠ NULL then
9         return HTree;
10    endif
11 endfor
12 return NULL;
```

—`cover` selects hyperedges for the separator that are satisfying the first condition of the *check*-step in Algorithm 2; we will describe it later in more detail.

—`expand` is used to complete a "truncated" hypertree decomposition: if the same hypertree node is constructed the second time during our search procedure, we know already if the corresponding components can be decomposed. Thus, we have to truncate the hypertree at this node in order to guarantee the polynomial runtime as mentioned above. The procedure `expand` expands such nodes to subtrees after the search process, which can be done in polynomial time.

We are now going to explain the transformation from k-`decomp` to `det-`k`-decomp` in more detail. The main procedures in Algorithm 1 and Algorithm 3 are almost the same. The only differences are the instantiation of the sets `FailSeps` and `SuccSeps`, which are used to store already visited separators, and the call of `expand`. If a component was successfully decomposed, the corresponding separator together with the component is inserted into `SuccSeps`, otherwise it is inserted into `FailSeps`. If a new separator is constructed during the search, it is checked whether it occurs in one of these two sets before its components are attempted to be decomposed. This avoids that the same component is decomposed multiple times. Intuitively, since there are at most n^k

Algorithm 5 *decompAdd* (*Edges*, *Conn*, *CovSep*)

```
 1 InCovSep := CovSep ∩ Edges;
 2 if InCovSep ≠ ∅ or k − |CovSep| > 0 then
 3     if InCovSep = ∅ then AddSize := 1 else AddSize := 0 endif;
 4     for each AddSep ⊆ Edges s.t. |AddSep| = AddSize do
 5         Separator := CovSep ∪ AddSep;
 6         Components := separate (Edges, Separator);
 7         if ∀Comp ∈ Components. ⟨Separator, Comp⟩ ∉ FailSeps then
 8             Subtrees := decompSub (Components, Separator);
 9             if Subtrees ≠ ∅ then
10                 Chi := Conn ∪ ⋃ (InCovSep ∪ AddSep);
11                 HTree := getHTNode (Separator, Chi, Subtrees);
12                 return HTree;
13             endif
14         endif
15     endfor
16 endif
17 return NULL;
```

Algorithm 6 *decompSub* (*Components*, *Separator*)

```
 1 Subtrees := ∅;
 2 for each Comp ∈ Components do
 3     ChildConn := ⋃Comp ∩ ⋃Separator;
 4     if ⟨Separator, Comp⟩ ∈ SuccSeps then
 5         HTree := getHTNode(Comp, ChildConn, ∅);
 6     else
 7         HTree := decompCov(Comp, ChildConn);
 8         if HTree = NULL then
 9             FailSeps := FailSeps ∪ {⟨Separator, Comp⟩};
10             return ∅;
11         else
12             SuccSeps := SuccSeps ∪ {⟨Separator, Comp⟩};
13         endif
14     endif
15     Subtrees := Subtrees ∪ {HTree};
16 endfor
17 return Subtrees;
```

separators each of size, at most, k and each with at most $n = |edges(HGraph)|$ components, the search procedure runs in polynomial time for fixed k.

Now, let us consider the recursive part in Algorithms 4–6. First, note that the second parameter of `decompCov` is a set of vertices instead of a set of hyperedges in k-`decomposable`. This is just a simplification, since `OldSep` in Algorithm 2 occurs only in the expression $\bigcup$`Edges` $\cap \bigcup$`OldSep`. Thus, we compute this set of "connecting vertices" `Conn` before the recursive call and replace `OldSep` by `Conn`.

The first few lines in Algorithm 4 are a simple optimization: if the current component contains, at most, k hyperedges, it can be trivially decomposed into a single hypertree-node. Afterward, in line 5, our determinization of the *guess and check*-part in Algorithm 2 starts. In particular, we compute the set

BoundEdges of hyperedges that are necessary to satisfy the condition in line 3 of Algorithm 2, which is Conn $\subseteq \bigcup$ Separator. To this aim it suffices to consider hyperedges in BoundEdges for the separator, i.e., hyperedges containing some vertices in Conn. Thus, we can reduce the search space for an appropriate separator by first selecting a subset CovSep of, at most, k hyperedges in BoundEdges such that Conn $\subseteq \bigcup$ CovSep, where CovSep $\subseteq$ Separator. This selection is done by the procedure cover in line 6. In particular, cover successively returns all possible selections and we have to loop through all of them. The further decomposition steps are then performed by decompAdd, which is called in line 7. If one such decomposition is successful, the resulting hypertree is returned in line 9; otherwise, NULL is returned in line 12 after all selections for CovSep have been tried. Note that the ordering of choosing CovSep is crucial for the duration of the search process in the case of success. Thus, we use heuristics in cover to obtain an appropriate selection ordering. We will explain these heuristics later in more detail.

Let us now consider the procedure decompAdd defined by Algorithm 5. Its third parameter is the set CovSep, which contains, at most, k hyperedges. These hyperedges were selected in such a way that the condition in line 3 of Algorithm 2 is satisfied. Since every separator guessed in k-decomposable must satisfy this condition and we loop through all possibilities for CovSep, we know that our search space is large enough to find all separators which are accepted by k-decomposable. This, however, is not necessary. To improve the runtime of our algorithm, we can restrict the search space based on the following insights.

Definition 1. [Gottlob et al. 2002] A hypertree decomposition of a hypergraph is in *normal form*, if for each $r \in vertices(T)$ and for each child s of r the following conditions hold:

1. there is (exactly) one $[\chi(r)]$-component C_r such that $\chi(T_s) = C_r \cup (\chi(s) \cap \chi(r))$,
2. $\chi(s) \cap C_r \neq \emptyset$, where C_r is the $[\chi(r)]$-component satisfying Condition 1, and
3. $\bigcup \lambda(s) \cap \chi(r) \subseteq \chi(s)$.

Definition 2. Let (T, χ, λ) be a hypertree decomposition of a hypergraph in normal form and $s \in vertices(T)$. We say s is *minimally labeled* if (i) $s = root(T)$ and $|\lambda(s)| = 1$ or (ii) s has parent r (with $[\chi(r)]$-component C_r such that $\chi(T_s) = C_r \cup (\chi(s) \cap \chi(r))$) and it holds that for each $e \in \lambda(s)$:

1. $\bigcup edges(C_r) \cap \bigcup \lambda(r) \not\subseteq \bigcup(\lambda(s) \setminus \{e\})$ or (Connectivity)
2. $(\lambda(s) \setminus \{e\}) \cap edges(C_r) = \emptyset$. (Monotonicity)

Definition 2 means that removing any edge from the λ-labels violates the connectivity or the monotonicity condition.

Definition 3. A hypertree decomposition of a hypergraph is in *strong normal form* if it is in normal form and each node is minimally labeled.

LEMMA 1. *For each k-width hypertree decomposition of a hypergraph H, there exists a k-width hypertree decomposition of H in strong normal form.*

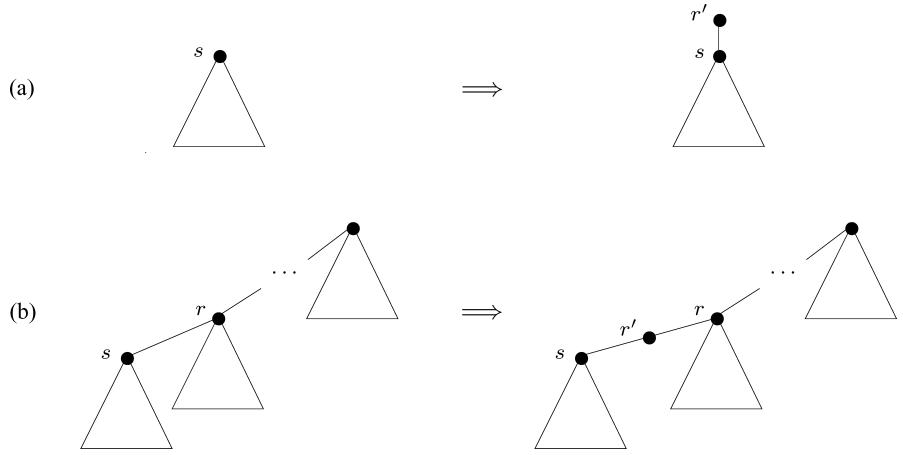


Fig. 3. Transformation steps to strong normal form.

PROOF. First note that we can assume w.l.o.g. that our hypertree decomposition (T, χ, λ) is in normal form [Gottlob et al. 2002]. We will show the lemma by an inductive argument on the size of the components. To this aim, let $s \in \mathit{vertices}(T)$ be not minimally labeled and assume that all other nodes on the path from s to $\mathit{root}(T)$ are minimally labeled. Now, we distinguish between two cases as illustrated in Figure 3: (a) If $s = \mathit{root}(T)$, we create a new node r' and add it to T as parent of s. We set $\lambda(r') = \{e\}$ for some hyperedge $e \in \lambda(s)$. (b) Otherwise, if $s \neq \mathit{root}(T)$, let r be the parent of s. We create a new node r' and add it to T between s and r, that is, r' is the parent of s and r is the parent of r'. We set $\lambda(r') = \lambda(s)$ and remove hyperedges from $\lambda(r')$ until any further removal would violate one of the conditions in Definition 2. Finally, we set $\chi(r') = \chi(s) \cap \bigcup \lambda(r')$ and remove all $v \in \chi(r) \setminus \chi(r')$ from $\chi(s')$ for each $s' \in \mathit{vertices}(T_s)$. It is easy to see that the resulting tree is still a hypertree decomposition, however, it is not necessarily in normal form. Since our modifications affect only the subtree rooted at r', we know that this subtree can be transformed into normal form according to Gottlob et al. [2002] without changing the labels of r'. Since $C_{r'} \subset C_r$ by the monotonicity condition, our inductive argument applies. □

Consequently, it suffices to consider the smallest separators satisfying the conditions in Algorithm 2. In particular, for `decompAdd` in Algorithm 5, this implies that it suffices to add, at most, one hyperedge to `CovSep` such that the condition in line 4 of Algorithm 2 is satisfied. To this aim, `decompAdd` computes in line 1 the intersection between `CovSep` and `Edges`. In line 2 it is then checked if the condition in line 4 of Algorithm 2 can be satisfied at all. In particular, if `InCovSep` is empty then we must add an edge to satisfy the condition. However, if `CovSep` contains already k hyperedges, this is not possible and we have to reject `CovSep`. Otherwise, in line 3 we set `AddSize` to 1 if we must add an edge and to 0 if `CovSep` already satisfies the condition. Then, in line 4, we loop through all hyperedges in `Edges` if `AddSize` $= 1$, that is, if we must add an edge; otherwise, `AddSep` is the empty set and the loop body is executed only once.

Finally, our separator is computed in line 5 as the union of `CovSep` and `AddSep`, where `AddSep` contains either a single hyperedge or is empty. In the latter case, `CovSep` satisfies both conditions in Algorithm 2 (recall that it satisfies the first condition by construction). Based on Lemma 1, we know that our computation of `Separator` is a correct determinization of the *guess and check*-part in Algorithm 2.

The call of `separate` in line 6 of Algorithms 2 and 5 is now completely the same in both algorithms. Since each subcomponent returned by `separate` must be decomposable, we check in line 7 of Algorithm 5 if one of these subcomponents is already known to be undecomposable. If so, we abort and choose an alternative separator. Otherwise, we try to decompose the subcomponents, which is done in the procedure `decompSub` in line 8. The resulting hypertree decompositions are returned in `Subtrees` if the decomposition was successful for all subcomponents; otherwise, `Subtrees` $= \emptyset$. Note that there must be at least one subcomponent in the case of success, i.e., `Subtrees` cannot be empty. Otherwise, a trivial decomposition of the current component would have been computed in lines 1– 4 of Algorithm 4. The construction of a new hypertree node is then performed in lines 10 and 11 completely analogous to lines 16 and 17 of Algorithm 2.

Finally, let us consider the procedure `decompSub` defined in Algorithm 6, which contains the functionality of lines 8–15 of Algorithm 2. In line 2, we loop through all components that have to be decomposed and in line 3 we compute the set of vertices which will be the second parameter `Conn` of the recursive call of `decompCov`. In order to guarantee the polynomial runtime, we have then to check in line 4 if the current subcomponent is already known to be decomposable. If so, we truncate the hypertree in line 5; otherwise, we apply the recursive call in line 7 analogous to line 9 in Algorithm 2. Depending on the result, i.e., whether the subcomponent could be decomposed or not, we store it either in `FailSeps` or in `SuccSeps`. Moreover, the constructed hypertree decompositions are stored in `Subtrees` in line 15 analogous to line 13 in Algorithm 2.

Thus, we have shown that our algorithm `det-`k`-decomp` is a deterministic variant of k`-decomp` defined in Algorithms 1 and 2. This insight of the correctness of `det-`k`-decomp` is now formulated below:

LEMMA 2. *For any given hypergraph H such that $hw(H) \leq k$,* `det-`k`-decomp` *accepts H. Moreover, for any $c \leq k$, each c-width hypertree decomposition of H in strong normal form is equal to some witness tree for H.*

PROOF. Let $HD = (T, \chi, \lambda)$ be a c-width hypertree decomposition of a hypergraph H in strong normal form, where $c \leq k$. We show that there exists an accepting computation tree for `det-`k`-decomp` on input H which coincides with HD. To this aim, note that by Lemma 9 in [Gottlob et al. 2002], there exists an accepting computation tree for k`-decomp` on input H which coincides with HD. Thus, it suffices to show that every accepting computation tree for k`-decomp` with corresponding hypertree decomposition in strong normal form is also an accepting computation tree for `det-`k`-decomp`. This, however, is trivial since `det-`k`-decomp` tries all computation trees of k`-decomp` whose corresponding hypertree decomposition is in strong normal form. □

LEMMA 3. *If* det-k-decomp *accepts a hypergraph* H*, then* $hw(H) \leq k$*. Moreover, each witness tree for* H *is a* c*-width hypertree decomposition of* H *in normal form, where* $c \leq k$*.*

PROOF. We show that every accepting computation tree for det-k-decomp on hypergraph H coincides with some c-width hypertree decomposition of H in normal form, where $c \leq k$. To this aim, note that by Lemma 13 in [Gottlob et al. 2002] every accepting computation tree for k-decomp on input H coincides with some c-width hypertree decomposition of H in normal form, where $c \leq k$. Thus, it suffices to show that every accepting computation tree for det-k-decomp is also an accepting computation tree for k-decomp. This, however, is trivial, since det-k-decomp tries a subset of all computation trees of k-decomp. □

Thus by combining Lemma 2 and Lemma 3 we get:

THEOREM 1. det-k-decomp *accepts a hypergraph* H *if and only if* $hw(H) \leq k$*.*

In the following we consider the runtime and space requirements of det-k-decomp. For simplicity, we assume a computation model where each elementary computation step needs a single time unit and each data word counts as a single space unit. Moreover, in analogy to Leone et al. [2002], we assume that the standard set operations can be performed in constant time. For better comparability, we ignore the auxiliary procedure expand since it is not necessary for the decision problem.

THEOREM 2. det-k-decomp *runs in time* $\mathcal{O}(n^k \min(nd^k, n^k) \min(m, n)^2)$*, where* m *and* n *are the number of vertices and the number of hyperedges in the hypergraph, respectively, and* d *is the maximum number of incident hyperedges over all hyperedges.*

PROOF. First note that there are, at most,

$$\Psi = \sum_{i=1}^{k} \binom{n}{i} = \sum_{i=1}^{k} \frac{n!}{i!(n-i)!}$$

separators each with at most $\min(m, n)$ subcomponents. Thus, since we restrict the number of recursive calls by checking SuccSeps and FailSeps to the number of possible separators and subcomponents, the number of recursive calls is bounded by $\mathcal{O}(\Psi \min(m, n))$. What remains to show is the runtime of a single recursive call. To this aim, note that the number of loops in line 6 of Algorithm 4 is, at most,

$$\sum_{i=1}^{k} \binom{\min(dk, n)}{i} = \sum_{i=1}^{k} \frac{\min(dk, n)!}{i!(\min(dk, n) - i)!},$$

which can be bounded by $\mathcal{O}(\min(d, n)^k)$. Since the number of loops in line 4 of Algorithm 5 is (at most) n only if less than k hyperedges have been chosen in line 6 of Algorithm 4, we can bound the number of loops in both algorithms by $\mathcal{O}(\min(nd^k, n^k))$. Moreover, the number of loops in line 2 of Algorithm 6 is, at most, $\min(m, n)$. Thus, the runtime of a single recursive call is bounded by $\mathcal{O}(\min(nd^k, n^k) \min(m, n))$. The overall time complexity

is, therefore, $\mathcal{O}(\Psi \min(nd^k, n^k) \min(m,n)^2)$, which, in turn, is bounded by $\mathcal{O}(n^k \min(nd^k, n^k) \min(m,n)^2)$. □

Since we can assume that d is in many cases much smaller than n, we have shown that det-k-decomp has a better runtime complexity than opt-k-decomp, which runs in time $\mathcal{O}(n^{2k}\, m^2)$ [Leone et al. 2002]. Moreover, note that similar to opt-k-decomp, the runtime may be significantly smaller in practice.

THEOREM 3. det-k-decomp *runs in space* $\mathcal{O}(n^k + \min(m,n)(m+n))$*, where* m *and* n *are the number of vertices and the number of hyperedges in the hypergraph, respectively.*

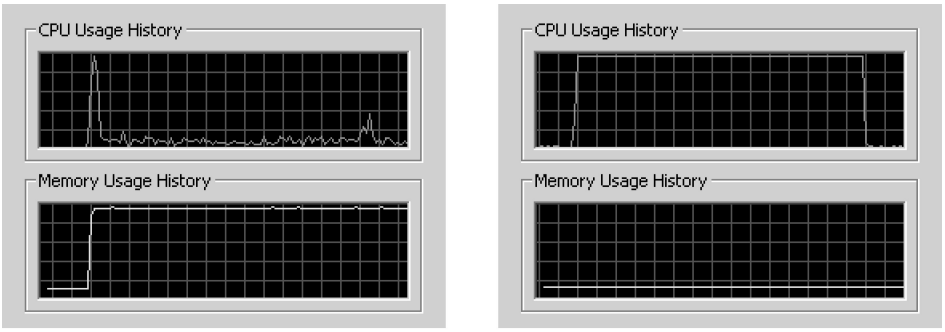
PROOF. Similar to the proof of Theorem 2, there are, at most,

$$\Psi = \sum_{i=1}^{k} \binom{n}{i} = \sum_{i=1}^{k} \frac{n!}{i!(n-i)!}$$

separators each with, at most, $\min(m,n)$ subcomponents. Thus, the space required for SuccSep and FailSep is bounded by $\mathcal{O}(\Psi)$. Since the number of nodes of each hypertree decomposition is bounded by $\min(m,n)$, the space requirement for a single node of the computation tree remains to be shown. Because of the loop in line 6 of Algorithm 4, we have to store Edges, Conn, BoundEdges, and CovSep, which can be done in space $\mathcal{O}(m+n)$. Moreover, because of the loop in line 4 of Algorithm 5, we have to store, at most, one hyperedge AddSep, which can be done in space $\mathcal{O}(1)$. Finally, because of the loop in line 2 of Algorithm 6, we have to store the components Comp already considered, which can be done in space $\mathcal{O}(\min(m,n))$. Thus, the space requirement for a single node of the computation tree is bounded by $\mathcal{O}(m+n)$. The overall space complexity is, therefore, $\mathcal{O}(\Psi + \min(m,n)(m+n))$, which, in turn, is bounded by $\mathcal{O}(n^k + \min(m,n)(m+n))$. □

Our empirical results show that the memory usage of det-k-decomp is often far below the memory usage of opt-k-decomp. For example, the CPU and memory usage of both algorithms when applied to instance s298 of the ISCAS89 benchmark suite can be seen in the screenshot of the Microsoft Windows Task Manager in Figure 4. Instance s298 represents a simple circuit hypergraph consisting of 139 vertices and 133 hyperedges. In Figure 4a, we see that the memory usage suddenly increases to its maximum when applying opt-k-decomp; therefore, the system resources are busy with memory swapping such that there are only minimal resources available for solving our decomposition problem. After an hour opt-k-decomp was still trying to solve the problem without success. In contrast, in Figure 4b, there is an unrecognizably small usage of memory when applying det-k-decomp; therefore, the full CPU power can be used for the decomposition problem, which was successfully solved within 90 seconds. We present more experimental results in Section 5, which demonstrate that det-k-decomp significantly outperforms opt-k-decomp.

Now, let us come back to our heuristics in the auxiliary procedure cover. As we mentioned above, the ordering of choosing CovSep in line 6 of Algorithm 4 has a crucial influence on the performance of det-k-decomp. We obtained our



(a) opt-k-decomp (b) det-k-decomp

Fig. 4. CPU and memory usage when decomposing s298 of the ISCAS89 benchmark suite.

Fig. 5. Covering $Conn = \{a, b, e\}$ by $BoundEdges = \{A, B, D, F, G, H\}$.

results with a very simple heuristic approach exemplified in Figure 5 by a decomposition step when decomposing the hypergraph in Figure 1. Let us assume that `Conn` is given by $\{a, b, e\}$ and `BoundEdges` is given by $\{A, B, D, F, G, H\}$. The task of the procedure `cover` is now to compute subsets of `BoundEdges` that cover `Conn` such that at least all minimal covers are considered. Recall that the minimal covers are sufficient because of Lemma 1. We do this in the following way: at first we assign to each hyperedge in `BoundEdges` a weight that is the number of vertices in `Conn` it contains. This can be seen on the left of Figure 5 where the weight of A is 2 since $A \cap \texttt{Conn} = \{a, b\}$, the weight of B is 1 since $B \cap \texttt{Conn} = \{e\}$, etc. Then we order the hyperedges according to their weight as it can be seen in the last line on the left of Figure 5. Finally, we choose the separators based on this ordering in the following way: we consider the hyperedges from left to right and select a hyperedge which covers a vertex in `Conn` that is not covered by other hyperedges selected so far. On the right in Figure 5, the selection of the five separators $\{A, F\}$, $\{A, B\}$, $\{A, G\}$, $\{F, D\}$, and $\{F, H\}$ is shown. For example, the first one is selected by choosing A, which covers a and b, and F, which covers e; the second one is selected by choosing B instead of F to cover e; etc. All five separators are computed in this way by choosing covering hyperedges from the left to the right until all vertices in `Conn` are covered. The first separator returned by `cover` is $\{A, F\}$, the second one is $\{A, B\}$, etc. A complete procedure call sequence, based on the above greedy heuristic, and the corresponding decomposition of the hypergraph in Figure 1 is shown in Figure 6.

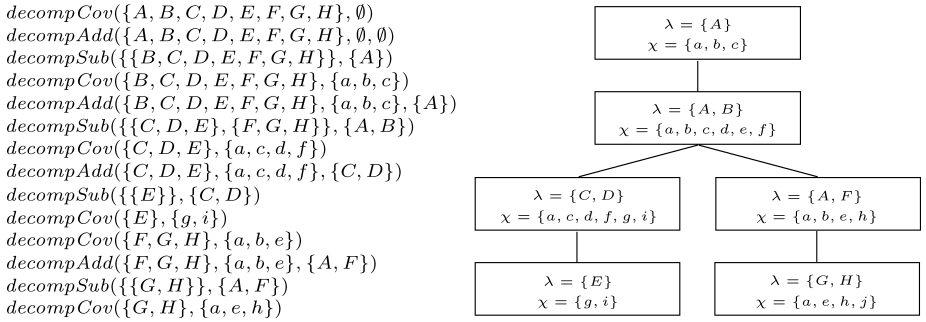


Fig. 6. Example of a procedure call sequence and the corresponding hypertree decomposition.

In addition to this heuristic approach, we use also some kind of randomization in our algorithm. In particular, the hyperedges read from the input file are randomly ordered. This guarantees that the results are independent from a particular representation of the hypergraph in the input file. However, this also means that `det-k-decomp` has to be applied several times for evaluation purposes since the results and execution times may differ in each run. Of course, this randomization step can be removed to take the input ordering of hyperedges and vertices into account, which sometimes improves the results.

5. EXPERIMENTAL RESULTS

In this section, we present our experimental results of the algorithm `det-k-decomp` when applied to three different classes of hypergraph instances. We compare these results with the results obtained from evaluations of `opt-k-decomp` (which can be downloaded from `http://www.deis.unical.it/scarcello/Hypertrees/`) and McMahan's heuristic approach. The latter one is based on *Bucket Elimination (BE)* with ordering heuristics *maximum cardinality*, *minimum induced-width* (also known as *minimum degree*), and *minimum fill-in* [McMahan 2004] (cf., Koster et al. [2001], Dechter [2003], Bodlaender [2005]). Our evaluations were performed on two different machines, since `opt-k-decomp` is only available as Microsoft Windows executable. Thus `opt-k-decomp` was evaluated under Microsoft Windows 2000 on a 2.4-GHz Intel Pentium 4 processor with 512-MB main memory and both `det-k-decomp` and `BE` were evaluated under SuSe Linux 9.2 on a 2.2-GHz Intel Xeon processor (dual) with 2-GB main memory. Note that the different memory sizes have no relevant influence on the results, since the memory usage of `det-k-decomp` and `BE` is far less than 512 MB when applied to our instances. We chose the Intel Xeon as reference machine for normalizing the execution times.

Since `BE` and `det-k-decomp` use some kind of randomization, we applied them five times to each example in order to obtain representative results. From these five runs, we selected the minimal width and computed the average runtime. Moreover, we defined a timeout of 3600 seconds for each run. Since we consider the width as the most important evaluation criterion and not the runtime, we set k always to the smallest value (less than or equal to the value obtained from `BE`) for which we obtained a solution from `det-k-decomp`

Table I. DaimlerChrysler Benchmarks

		opt-k-decomp		BE		det-k-decomp	
Instance (Vertices/Edges)	Min	*width*	*time*	*width*	*time*	*width*	*time*
adder_15 (106/76)	2	**2**	5	**2**	0	**2**	0
adder_25 (176/126)	2	**2**	40	**2**	0	**2**	0
adder_50 (351/251)	2	—	—	**2**	0	**2**	0
adder_75 (526/376)	2	—	—	**2**	0	**2**	0
adder_99 (694/496)	2	—	—	**2**	1	**2**	0
bridge_15 (137/137)	2	**2**	19	**3**	0	**2**	0
bridge_25 (227/227)	2	**2**	138	**3**	0	**2**	0
bridge_50 (452/452)	2	**2**	2211	**3**	1	**2**	0
bridge_75 (677/677)	2	—	—	**3**	1	**2**	0
bridge_99 (893/893)	2	—	—	**3**	2	**2**	1
NewSystem1 (142/84)	3	—	—	**3**	0	**3**	0
NewSystem2 (345/200)	3	—	—	**4**	0	**3**	0
NewSystem3 (474/278)	—	—	—	**5**	1	**4**	0
NewSystem4 (718/418)	—	—	—	**5**	2	**4**	0
atv_partial_system (125/88)	3	—	—	**3**	0	**3**	0

within our timeout. Although not done in our evaluations, note that in principle we can remove the parameter k from `det-`k`-decomp` by defining `detdecomp` as the algorithm obtained by looping through all k from $k = \lceil n/2 \rceil$ to $k = 1$ and returning the smallest k for which `det-`k`-decomp` terminates within the timeout, where n is the number of hyperedges in the given hypergraph. Note, however, that such an algorithm `detdecomp` does no longer run in polynomial time since k is not a constant anymore.

The first class of our benchmarks are from DaimlerChrysler and consist of hypergraphs extracted from adder and bridge circuits, the NewSystem examples, and a model of a jet propulsion system [Korimort 2003; Ganzow et al. 2005]. The results when applying our three algorithms to theses hypergraphs are shown in Table I. The first row in each line contains the name of the example and its size, i.e., the number of vertices and the number of hyperedges. The second row contains the hypertree-width, if known, i.e., the optimal result. Then there are two rows for each algorithm; the first one contains the minimal width and the second one the average runtime in seconds. It is easy to see that `det-`k`-decomp` outperforms both k`-decomp` and `BE` on the DaimlerChrysler examples.

The second class of benchmarks are hypergraphs extracted from two-dimensional grids [Ganzow et al. 2005]. These instances are interesting since their hypertree-width is known by construction. Although we can easily construct an optimal decomposition of these examples by hand, it is seemingly very hard for our algorithms, as can be seen in Table II. Note that `opt-`k`-decomp` cannot solve any of them within our timeout of one hour. Moreover, note that the runtime of `det-`k`-decomp` is very high for larger examples. One reason for this may be that we set k to the smallest value for which we obtain a decomposition within an hour. If we increase k, then the runtime may decrease in many cases.

Finally, the third class of benchmarks are hypergraphs extracted from circuits of the ISCAS89 (International Symposium on Circuits and Systems)

Table II. Grid2D Benchmarks

		opt-k-decomp		BE		det-k-decomp	
Instance (Vertices/Edges)	Min	*width*	*time*	*width*	*time*	*width*	*time*
grid2d_10 (50/50)	4	—	—	**5**	0	**4**	0
grid2d_15 (113/112)	6	—	—	**8**	0	**6**	3
grid2d_20 (200/200)	7	—	—	**12**	0	**7**	3140
grid2d_25 (313/312)	9	—	—	**15**	3	**10**	2000
grid2d_30 (450/450)	11	—	—	**19**	7	**13**	1566
grid2d_35 (613/612)	12	—	—	**23**	15	**15**	1905
grid2d_40 (800/800)	14	—	—	**26**	28	**17**	2530
grid2d_45 (1013/1012)	16	—	—	**31**	51	**21**	2606
grid2d_50 (1250/1250)	17	—	—	**33**	86	**24**	2786
grid2d_60 (1800/1800)	21	—	—	**41**	204	**31**	2984
grid2d_70 (2450/2450)	24	—	—	**48**	474	**42**	2161
grid2d_75 (2813/2812)	26	—	—	**48**	631	**45**	2881

Table III. ISCAS89 Benchmarks

		opt-k-decomp		BE		det-k-decomp	
Instance (Vertices/Edges)	Min	*width*	*time*	*width*	*time*	*width*	*time*
s27 (17/13)	2	**2**	0	**2**	0	**2**	0
s208 (115/104)	$\geq$3	—	—	**7**	0	**6**	0
s298 (139/133)	$\geq$3	—	—	**5**	0	**4**	462
s344 (184/175)	$\geq$3	—	—	**7**	0	**5**	730
s349 (185/176)	$\geq$3	—	—	**7**	0	**5**	4
s382 (182/179)	$\geq$3	—	—	**5**	0	**5**	722
s386 (172/165)	—	—	—	**8**	1	**7**	1824
s400 (186/183)	$\geq$3	—	—	**6**	0	**5**	273
s420 (231/212)	$\geq$3	—	—	**9**	0	**8**	454
s444 (205/202)	$\geq$3	—	—	**6**	0	**5**	385
s510 (236/217)	$\geq$3	—	—	**23**	1	**20**	2082
s526 (217/214)	$\geq$3	—	—	**8**	1	**7**	1715
s641 (433/398)	—	—	—	**7**	1	**7**	1611
s713 (447/412)	—	—	—	**7**	1	**7**	1800
s820 (312/294)	$\geq$3	—	—	**13**	3	**12**	2846
s832 (310/292)	$\geq$3	—	—	**12**	3	**11**	2575
s838 (457/422)	$\geq$3	—	—	**16**	1	**15**	2046
s953 (440/424)	$\geq$3	—	—	**40**	8	—	—
s1196 (561/547)	—	—	—	**35**	11	—	—
s1238 (540/526)	—	—	—	**34**	13	—	—
s1423 (748/731)	—	—	—	**18**	3	—	—
s1488 (667/659)	—	—	—	**23**	18	—	—
s1494 (661/653)	—	—	—	**24**	19	—	—
s5378 (2993/2958)	—	—	—	**85**	141	—	—

benchmark suite. The ISCAS benchmarks are again examples from industry. However, they are obviously much more difficult to solve than the Daimler-Chrysler instances, as can be seen in Table III. Note that the widths obtained from `det-`k`-decomp` are not much smaller than those from `BE`, although it needs much more time.

In summary, we can conclude that `det-`k`-decomp` significantly outperforms `opt–`k`-decomp` on all classes of benchmarks we have tested. `opt-`k`-decomp`

terminates within our timeout only for very small and simple examples. In contrast, det-k-decomp terminates also for larger examples with results comparable to or even better than those computed by BE. Thus, if the width is the most important criterion and not the runtime, det-k-decomp also outperforms BE, as long as the hypergraphs are sufficiently small and simple.

6. CONCLUSION

We have presented the new algorithm det-k-decomp for constructing hypertree decompositions of hypergraphs. This algorithm results from the alternating algorithm k-decomp introduced by Gottlob et al. [2002] when replacing the *guess and check*-part by a search procedure. We have evaluated det-k-decomp by a series of benchmark examples from industry and academics and compared it to the currently best hypertree decomposition algorithms. Our results have shown that det-k-decomp performs very well; in particular, it significantly outperforms the algorithm opt-k-decomp. We hope that our work stimulates further research in this area that helps to improve the performance of det-k-decomp. This could be theoretical results to restrict the search space, experimental results to identify more sophisticated heuristics that accelerate the search process, and more efficient implementations of the whole algorithm. We believe that there is a lot of potential for such improvements.

A. APPENDIX

PROPOSITION 4. *The algorithm* k-decomp *presented in Algorithms* 1 *and* 2 *is a notational variant of the algorithm* k-decomp *presented in* [*Gottlob et al.* 2002].

PROOF. First note that k-decomp returns NULL if it rejects and it returns a c-width hypertree decomposition if it accepts, where $c \leq k$. Algorithm 1 represents the main procedure, which calls the recursive procedure k-decomposable described in Algorithm 2. The names of these procedures are the same as in their original presentation by Gottlob et al. [2002]. The first parameter of k-decomposable, however, is a set of edges instead of a set of vertices. In particular, we use the set $edges(C_R)$ of hyperedges containing some vertices in the component C_R instead of the component C_R itself. Both variants are equivalent, since $C_R = \bigcup edges(C_R) \setminus \bigcup R$ can be computed from $edges(C_R)$ and R. Thus, the first parameter of k-decomposable is the set of hyperedges representing the current component and the second parameter is the set of λ-labels of the parent node. Note that we use the term *separator* for a set of hyperedges chosen as λ-labels, since they separate the subcomponents.

Now, let us compare Algorithm 2 with its original formulation by Gottlob et al. [2002]. The *guess*-step of Separator in line 1 is obviously the same. For the *check*-step in lines 2–4, it is easy to see that both conditions are equivalent to their original formulation. Line 6 describes the computation of the set $\mathcal{C}$ of components in [Gottlob et al. 2002]. This computation is now hidden in the procedure separate and its result Components is a set of sets of hyperedges such that for each $C \in \mathcal{C}$ there exists Comp $\in$ Components, such that $\bigcup$Comp $\setminus$ $\bigcup$Separator $= C$. The remainder of Algorithm 2 contains the recursive call as

in [Gottlob et al. 2002] and the construction of the hypertree decomposition. In particular, the recursive call itself is given in line 9, where we loop through all components. If the decomposition of a component is rejected, we reject in line 11; otherwise, we store the hypertree decomposition of the component in `Subtrees` in line 13. Finally, a new hypertree node is constructed in line 17 with `Separator` as λ-labels and `Subtrees` as subtrees; the χ-labels are computed according to line 16. Note that the χ-labels computed in this way do not necessarily satisfy the third normal-form condition. However, this condition can be easily satisfied by a simple postprocessing step.

Thus, we have demonstrated the equivalence between our notational variant of the algorithm k-`decomp` and its original formulation in [Gottlob et al. 2002]. □

REFERENCES

BODLAENDER, H. L. 2005. Discovering treewidth. In *Proceedings of the 31st Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM'05)*. Lecture Notes in Computer Science, vol. 3381. Springer-Verlag, New York. 1–16.

DECHTER, R. 2003. *Constraint Processing*, Chapt. 4, Morgan Kaufmann, Burlington, MA. 85–115.

GANZOW, T., GOTTLOB, G., MUSLIU, N., AND SAMER, M. 2005. A CSP hypergraph library. Tech. Rept. DBAI-TR-2005-50, Institute of Information Systems (DBAI), TU Vienna.

GOTTLOB, G., LEONE, N., AND SCARCELLO, F. 1999. On tractable queries and constraints. In *Proceedings of 10th International Conference on Database and Expert System Applications (DEXA)*. Lecture Notes in Computer Science, vol. 1677. Springer-Verlag, New York. 1–15.

GOTTLOB, G., LEONE, N., AND SCARCELLO, F. 2002. Hypertree decompositions and tractable queries. *J. Comput. Syst. Sci. 64*, 3, 579–627.

GOTTLOB, G., LEONE, N., AND SCARCELLO, F. 2003. Robbers, marshals, and guards: Game theoretic and logical characterizations of hypertree width. *J. Comput. Syst. Sci. 66*, 4, 775–808.

GOTTLOB, G., MIKLÓS, Z., AND SCHWENTICK, T. 2007. Generalized hypertree decompositions: NP-hardness and tractable variants. In *Proceedings of the 26th ACM Symposium on Principles of Database Systems (PODS'07)*. ACM Press, New York. 13–22.

HARVEY, P. AND GHOSE, A. 2003. Reducing redundancy in the hypertree decomposition scheme. In *Proceedings of 15th International Conference on Tools with Artificial Intelligence (ICTAI'03)*. IEEE Computer Society, Los Alamitos, CA. 474–481.

KORIMORT, T. 2003. Constraint satisfaction problems—heuristic decomposition. Ph.D. thesis, Institute of Information Systems (DBAI), TU Vienna.

KOSTER, A. M. C. A., BODLAENDER, H. L., AND VAN HOESEL, S. P. M. 2001. Treewidth: Computational experiments. Tech. Rept. ZIB-Report 01-38, Konrad-Zuse-Zentrum für Informationstechnik Berlin (ZIB).

LEONE, N., MAZZITELLI, A., AND SCARCELLO, F. 2002. Cost-based query decompositions. In *Proceedings of the 10th Italian Symposium on Advanced Database Systems* (*SEBD*).

MCMAHAN, B. 2004. Bucket elimination and hypertree decompositions. Implementation Rept. Institute of Information Systems (DBAI), TU Vienna.

SCARCELLO, F., GRECO, G., AND LEONE, N. 2007. Weighted hypertree decompositions and optimal query plans. *J. Comput. Syst. Sci. 73*, 3, 475–506.

Received January 2007; revised October 2007; accepted March 2008