Hypertree Decompositions: Structure, Algorithms, and Applications*

Georg Gottlob[1], Martin Grohe[2], Nysret Musliu[1], Marko Samer[1], and Francesco Scarcello[3]

[1] Institut für Informationssysteme, TU Wien, Vienna, Austria
[2] Institut für Informatik, Humboldt-Universität, Berlin, Germany
[3] D.E.I.S., University of Calabria, Rende (CS), Italy

Abstract We review the concepts of hypertree decomposition and hypertree width from a graph theoretical perspective and report on a number of recent results related to these concepts. We also show – as a new result – that computing hypertree decompositions is fixed-parameter intractable.

1 Hypertree Decompositions: Definition and Basics

This paper reports about the recently introduced concept of *hypertree decomposition* and the associated notion of *hypertree-width*. The latter is a cyclicity measure for hypergraphs, and constitutes a hypergraph invariant as it is preserved under hypergraph isomorphisms. Many interesting NP-hard problems are polynomially solvable for classes of instances associated with hypergraphs of bounded width. This is also true for other hypergraph invariants such as treewidth, cutset-width, and so on. However, the advantage of hypertree-width with respect to other known hypergraph invariants is that it is more general and covers larger classes of instances of bounded width. The main concepts of hypertree decomposition and hypertree-width are introduced in the present section. A normal form for hypertree decompositions is described in Section 2. Section 3 describes the Robbers and Marshals game which characterizes hypertree-width. In Section 4 we use this game to explain why the problem of checking whether the hypertree-width of a hypergraph is $\leq k$ is feasible in polynomial time for each constant k. However, in Section 5 we show that this problem is fixed-parameter intractable with respect to k. In Section 6 we compare hypertree-width to other relevant hypergraph invariants. In Section 7 we discuss heuristics for computing hypertree decompositions. In Section 8 we show how hypertree decompositions can be beneficially applied for solving constraint satisfaction problems (CSPs). Finally, in Section 9 we list some open problems left for future research. Due to space limitations this paper is rather short, and most proofs are missing. A more thorough treatment can be found in [13,16,2,1,15,17], most of which are available at the Hypertree Decomposition Homepage at http://si.deis.unical.it/∼frank/Hypertrees.

* This paper was supported by the Austrian Science Fund (FWF) project: *Nr. P17222-N04, Complementary Approaches to Constraint Satisfaction*. Correspondence to: Georg Gottlob, Institut für Informationssysteme, TU Wien, Favoritenstr. 9-11/184-2, A-1040 Wien, Austria, E-mail: gottlob@acm.org.

A *hypergraph* is a pair $H = (V(H), E(H))$, consisting of a nonempty set $V(H)$ of *vertices*, and a set $E(H)$ of subsets of $V(H)$, the *hyperedges* of H. We only consider finite hypergraphs. *Graphs* are hypergraphs in which all hyperedges have two elements.

For a hypergraph H and a set $X \subseteq V(H)$, the *subhypergraph induced by* X is the hypergraph $H[X] = (X, \{e \cap X \mid e \in E(H)\})$. We let $H \setminus X := H[V(H) \setminus X]$. The *primal graph* of a hypergraph H is the graph

$$\underline{H} = (V(H), \{\{v, w\} \mid v \neq w, \text{ there exists an } e \in E(H) \text{ such that } \{v, w\} \subseteq e\}).$$

A hypergraph H is *connected* if $\underline{H}$ is connected. A set $C \subseteq V(H)$ is *connected (in H)* if the induced subhypergraph $H[C]$ is connected, and a *connected component* of H is a maximal connected subset of $V(H)$. A sequence of nodes of $V(H)$ is a *path* of H if it is a path of $\underline{H}$.

A *tree decomposition* of a hypergraph H is a tuple (T, χ), where $T = (V(T), E(T))$ is a tree and $\chi : V(T) \longrightarrow 2^{V(H)}$ is a function associating a set of vertices $\chi(t) \subseteq V(H)$ to each vertex t of the decomposition tree T, such that for each $e \in E(H)$ there is a node $t \in V(T)$ such that $e \subseteq \chi(t)$, and for each $v \in V(H)$ the set $\{t \in V(T) \mid v \in \chi(t)\}$ is connected in T.

We assume the tree T in a tree decomposition to be rooted. For every node t, T_t denotes the rooted subtree of T with root t. For each such subtree T_t, let $\chi(T_t) = \bigcup_{v \in V(T_t)} \chi(v)$.

The *width* of a tree decomposition (T, χ) is $\max\left\{\, |\chi(t)| - 1 \;\middle|\; t \in V(T)\right\}$, and the *tree-width* of H is the minimum of the widths of all tree decompositions of H.

Observe that (T, χ) is a tree decomposition of H if and only if it is a tree decomposition of $\underline{H}$. Thus a hypergraph has the same tree-width as its primal graph.

Let H be a hypergraph. A *generalized hypertree decomposition of H* is a triple (T, χ, λ), where (T, χ) is a tree decomposition of H and $\lambda : V(T) \longrightarrow 2^{E(H)}$ is a function associating a set of hyperedges $\lambda(t) \subseteq E(H)$ to each vertex t of the decomposition tree T, such that for every $t \in V(T)$ we have $\chi(t) \subseteq \bigcup \lambda(t)$. The *width* of a generalized hypertree decomposition (T, χ, λ) is $\min\{|\lambda(t)| \mid t \in V(T)\}$, and the *generalized hypertree-width* $\mathrm{ghw}(H)$ of H is the minimum of the widths of all generalized hypertree decompositions of H.

A *hypertree decomposition* of H is a generalized hypertree decomposition (T, χ, λ) that satisfies the following *special condition*: $(\bigcup \lambda(t)) \cap \chi(T_t) \subseteq \chi(t)$ for all $t \in V(T)$. The *hypertree-width* $\mathrm{hw}(H)$ of H is the minimum of the widths of all hypertree decompositions of H.

Example 1. Figure 1 shows a hypergraph H (consisting of 15 hyperedges and 19 vertices) and a tree decomposition of H. A generalized hypertree decomposition and a hypertree decomposition of H are illustrated in Figure 2. The left set within each rectangle represents the λ-labels and the right set represents the χ-labels. The generalized hypertree decomposition violates the special condition, because vertex 13 disappears from node with λ-label $\{h10, h14\}$ and it appears again in a subtree rooted at this node. The generalized hypertree-width of H is 2, whereas its hypertree-width is 3.

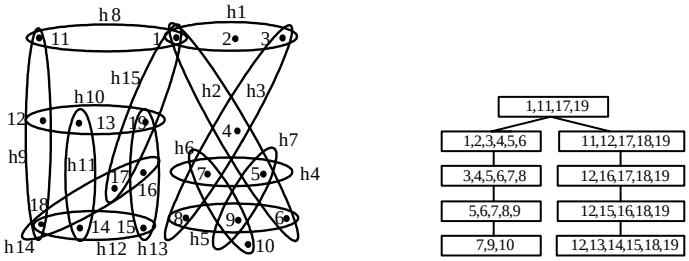

Figure 1. A hypergraph H (left) and a tree decomposition of H (right)

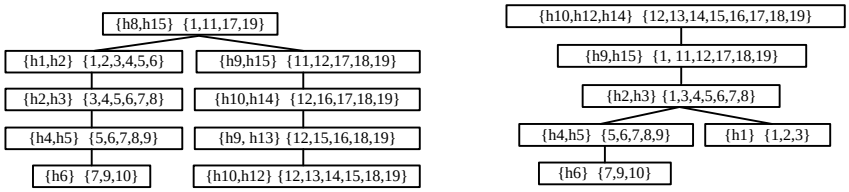

Figure 2. Generalized hypertree decomposition (left) and hypertree decomposition (right) of H

Example 2. Let H be a the hypergraph with $V(H) = \{1, \ldots, n\}$ and

$$E(H) = \big\{\{v, w\} \;\big|\; v, w \in V(H) \text{ with } v \neq w\big\} \cup \big\{V(H)\big\}.$$

Hence H is the hypergraph obtained from a complete graph with n vertices by adding a hyperedge that contains all vertices. It is easy to see that $\mathrm{hw}(H) = 1$ and $\mathrm{tw}(H) = n - 1$. Moreover, even the treewidth of the bipartite incidence graph of H is $n - 1$.

The structure of many problems can be described by hypergraphs (see also Section 8). Let us informally define a *hypergraph decomposition* as a method of dividing hypergraphs into different parts so that the solution of certain problems whose structure is best described by hypergraphs can be obtained by a polynomial divide-and-conquer algorithm that suitably exploits this division. The *width* of such a decomposition is the size of the largest indecomposable part of this division.

The importance of hypergraph decompositions (be it tree decompositions, hypertree decompositions, or several others) lies in the fact that many problems can be polynomially solved if their associated hypergraph has a low width for the chosen decomposition (see Section 8). The problem is thus to find decompositions that have the following properties:

1. They should be as *general* as possible, i.e., so that the classes of hypergraphs of bounded width are as large as possible. A criterion for comparing the generality of decomposition methods will be given in Section 6.
2. They should be *polynomially computable*. More precisely, for each fixed constant k, we want to be able to check in polynomial time whether a decomposition of width k of an input hypergraph exists.

3. Hypergraph decompositions of bounded width should lead to the polynomial solution of the underlying problem (e.g. of constraint satisfaction problems as described in Section 8). Typically, we expect that for a decomposition of a certain type, the class of problems whose associated hypergraph has width bounded by k can be solved in time $O(n^{O(k)})$.

Several decomposition methods satisfy properties 2 and 3, in particular the method of hypertree decomposition. Hypertree decompositions also satisfy Property 1. By results of [15] and [2], which will be briefly reviewed in Section 6, the method of hypertree decompositions is – so far – the most general method satisfying all three of the above criteria.

2 A Normal Form for Hypertree Decompositions

Let $H = (V(H), E(H))$ be a hypergraph, and let $V \subseteq V(H)$ be a set of vertices and $a, b \in V(H)$. Then a is $[V]$-adjacent to b if there exists an edge $h \in E(H)$ such that $\{a, b\} \subseteq h \setminus V$. A $[V]$-path π from a to b is a sequence $a = a_0, a_1, a_2, \ldots, a_\ell = b$ of vertices such that a_i is $[V]$-adjacent to a_{i+1}, for each $i \in [0, \ell - 1]$. A set $W \subseteq V(H)$ of vertices is $[V]$-connected if, for all $a, b \in W$, there is a $[V]$-path from a to b. A **$[V]$-*component*** is a maximal $[V]$-connected non-empty set of vertices $W \subseteq V(H) \setminus V$. For any set C of vertices, let $edges(C) = \{h \in E(H) \mid h \cap C \neq \emptyset\}$.

Let $HD = (T, \chi, \lambda)$ be a generalized hypertree decomposition for H. For any vertex $v \in V(T)$, we will often use v as a synonym of $\chi(v)$. In particular, **$[v]$-*component*** denotes **$[\chi(v)]$-*component***; the term $[v]$-path is a synonym of $[\chi(v)]$-path; and so on. We introduce a normal form for generalized hypertree decompositions, and thus also for hypertree decompositions.

Definition 1 ([13]). A generalized hypertree decomposition $HD = (T, \chi, \lambda)$ of a hypergraph H is in *normal form* (*NF*) if, for each vertex $r \in V(T)$, and for each child s of r, all the following conditions hold:
1. there is (exactly) one **$[r]$-*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 **$[r]$**-component satisfying Condition 1;
3. $(\bigcup \lambda(s)) \cap \chi(r) \subseteq \chi(s)$.

Intuitively, each subtree rooted at a child node s of some node r of a normal form decomposition tree serves to decompose precisely one **$[r]$-*component***.

Theorem 1 ([13]). *For each k-width hypertree decomposition of a hypergraph H there exists a k-width hypertree decomposition of H in normal form.*

3 Robbers and Marshals

In [29], graphs G of treewidth k are characterized by the so called *Robber-and-Cops game* where $k + 1$ cops have a winning strategy for capturing a robber on G. Cops can control vertices of a graph and can fly at each move to arbitrary vertices, say, by using a helicopter. The robber can move (at infinite speed) along paths of G, and will try to

escape the approaching helicopter(s), but cannot go over vertices controlled by a cop. It is, moreover, shown that a winning strategy for the cops exists, iff the cops can capture the robber in a *monotone* way, i.e., never returning to a vertex that a cop has previously vacated, which implies that the moving area of the robber is monotonic shrinking. For more detailed descriptions of the game, see [29] or [16].

In order to provide a similarly natural characterization for hypertree-width, a new game, the *Robber and Marshals game (R&Ms game)*, was defined in [16]. A marshal is more powerful than a cop. While a cop can control a single vertex of a hypergraph H only, a marshal controls an entire hyperedge. In the *R&Ms* game, the robber moves on vertices along a path of H (i.e., a path of the primal graph $\underline{H}$) just as in the robber and cops game, but now marshals instead of cops are chasing the robber. During a move of the marshals from the set of hyperedges E to the set of hyperedges E', the robber cannot pass through the vertices in $B = (\bigcup E) \cap (\bigcup E')$, where, for a set of hyperedges F, $\bigcup F$ denotes the union of all hyperedges in F. Intuitively, the vertices in B are those not released by the marshals during the move.

In this game, the set of all marshals is considered to be one player and the robber the other player. The marshals objective is thus to move a marshal (via helicopter) on a hyperedge containing the vertex occupied by the robber. The robber tries to elude capture. As for the robber and cops game, we distinguish between a *general* (not necessarily monotone) and a *monotone* version of the *R&Ms* game. In the monotone version of the game, the marshals have to make sure, that in each step the robber's escape space, i.e., the component in which the robber can freely move around, decreases. The *(monotone) marshal-width* of a hypergraph H, $\mathrm{mw}(H)$ (and $\text{mon-mw}(H)$, respectively), is the least number k of marshals that have a (monotone) winning strategy in the robber and k marshals game played on H (see [1,16] for more precise definitions).

Clearly, for each hypergraph H, $\mathrm{mw}(H) \leq \text{mon-mw}(H)$. However, unlike for the robber and cops game, the marshal width and the monotone marshal width differ. Adler [1] proved that for each constant k there is a hypergraph H such that $\text{mon-mw}(H) - \mathrm{mw}(H) = k$.

In [16] it is shown that there is a one-to-one correspondence between the winning strategies for k marshals in the monotone game and the normal-form hypertree decompositions of width at most k.

Theorem 2 ([16]). *A hypergraph H has k-bounded hypertree-width if and only if k marshals have a winning strategy for the monotone* R&Ms *game played on H.*

4 Computing Hypertree Decompositions

For each constant k it can be decided in polynomial time whether a given hypergraph H has a k-bounded hypertree decomposition. In this section we briefly sketch the algorithm k-`decomp` which solves this problem in logarithmic space via alternating computations.

The algorithm is best understood via the monotone *R&Ms* game. A typical game situation is depicted in Figure 3, where we assume that the marshals are at some instant in position R, i.e., occupy a set R of k hyperedges, and that the robber is in a component

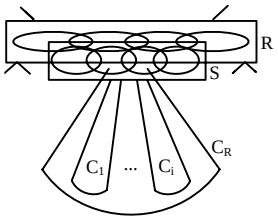

Figure 3. Marshals moving from position R to position S

C_R corresponding to this position of the marshals, i.e., in an **[**$\bigcup R$**]**-$component\ C_R$. In the next move, the marshals must chase the robber within C_R. They move to a new position $S \subseteq E(H)$ determined by at most k hyperedges. This move is a correct move in the game iff the following conditions are satisfied: (a) the robber cannot escape from component C_R during or after the move of the marshals, and (b) the escape space of the robber is effectively shrinking.

Condition (a) is mathematically expressed through the statement

$$(a)\ \ \forall P \in edges(C_R),\ (P \cap \bigcup R) \subseteq \bigcup S.$$

In fact, since C_R is an **[**$\bigcup R$**]**-component, all ways out of it pass through the set of vertices $\bigcup R$ of R. Thus, if there is a way out of C_R, there must be an edge P of $edges(C_R)$ leading from C_R into R. However, by the above condition (a), the robber cannot enter R through this edge P, because the set $P \cap \bigcup R$ of vertices of P that are in R are also in S and remain thus off-limits for the robber both during and after the move of the marshals.

Assuming that condition (a) is satisfied, it is easy to see that to make sure that the escape space has shrunk after the move of the marshals, it suffices to require that the new marshal position S covers at least one vertex of the former escape space C_R, formally:

$$(b)\ \ (\bigcup S) \cap C_R\ \neq \emptyset.$$

In fact, Condition (a) already guarantees that the escape space cannot become larger. Condition (b) requires that some vertex of the former escape space be covered by the marshals after the move, and thus the escape space must shrink. Notice that the original escape space C_R, after the move of the marshals from R to S may be split into several **[**$\bigcup S$**]**-components $C_1, C_2, \ldots, C_i, \ldots$

Figure 4 shows (a high-level description of) the algorithm k-`decomp`. This algorithm tries to construct a winning strategy for k marshals to win the *R&Ms* game on an input hypergraph H. Such a winning strategy is constructed in an alternating fashion by the procedure ***k-decomposable***(C_R, R) which has as parameters a marshals position R (i.e. a set of $\leq k$ hyperedges of H), and an **[**$\bigcup R$**]**-$component\ C_R$ which is the current escape space where the robber is to be chased. The procedure guesses (as an existential computation) in Step 1 a marshals position S, and checks in Steps 2.a and 2.b, whether

this position is correct according to the above discussed conditions (a) and (b), respectively. The algorithm then determines (in Step 3) the new components determined by the S-position of the marshals and recursively checks if the k marshals have a winning strategy for *each* of these components C by calling *k-decomposable*(C, S). The algorithm accepts if this is the case and rejects otherwise. This part is clearly a universal computation.

The algorithm is initialized (MAIN program) by the call *k-decomposable*$(V(H), \emptyset)$ where $V(H)$ is the initial escape space consisting of the entire vertex set of H, and where the initial marshals position is the empty set, i.e., where no hyperedge is occupied by a marshal. The correctness of the algorithm follows easily from the characterization of hypertree-width through the *R&Ms* game (Theorem 2). A direct proof (not involving the *R&Ms* game) is given in [16].

ALTERNATING ALGORITHM k-`decomp`
Input: A non-empty Hypergraph $H = (V(H), E(H))$.
Result: "Accept", if H has k-bounded hypertree-width; "Reject", otherwise.

Procedure *k-decomposable(*C_R*: SetOfVertices,* R*: SetOfHyperedges)*
begin
1) **Guess** a set $S \subseteq E(H)$ of k elements at most;
2) **Check** that all the following conditions hold:
   2.a) $\forall P \in edges(C_R),\ (P \cap \bigcup R) \subseteq \bigcup S$ and
   2.b) $(\bigcup S) \cap C_R \neq \emptyset$
3) **If** the check above fails **Then Halt** and **Reject**; **Else**
   Let $\mathcal{C} := \{C \subseteq V(H) \mid C\ is\ a$ **[**$\bigcup S$**]**-*component and* $C \subseteq C_R\}$;
4) **If, for each** $C \in \mathcal{C}$, *k-decomposable*(C, S)
   Then Accept
   Else Reject
end;

begin*(* MAIN *)*
   Accept if *k-decomposable*$(V(H), \emptyset)$
end.

Figure 4. *A non-deterministic algorithm deciding k-bounded hypertree-width*

A position U of k marshals can be stored as k pointers to (or indices of) hyperedges of H, and, each **[**$\bigcup U$**]**-component can be identified through a single vertex. Thus the workspace required at the global level of the initial and each recursive activation of k-`decomp` is logarithmic in the size of the input hypergraph H. Thus k-`decomp` can be implemented on an alternating Turing machine using logarithmic workspace, which proves that the associated decision problem is solvable in polynomial time. Actually, a witness of a successful computation corresponds to a hypertree decomposition in NF, thus k-`decomp` can actually be implemented on a logspace ATM having *polynomially bounded tree-size*, cf. [27], and therefore deciding whether $\mathrm{hw}(H) \leq k$ for a hypergraph H is actually in the low complexity class LOGCFL. This is the class of all

problems that are logspace-reducible to a context-free language. LOGCFL is a subclass of the class AC^1 of highly parallelizable problems.

Theorem 3 ([13]). *Deciding whether a hypergraph H has k-bounded hypertree-width is in* LOGCFL.

From an accepting computation of the algorithm of Figure 4 we can efficiently extract a NF hypertree decomposition. Since an accepting computation tree of a bounded-treesize logspace ATM can be *computed* in (the functional version of) LOGCFL [12], we obtain the following:

Theorem 4 ([13]). *Computing a k-bounded hypertree decomposition (if any) of a hypergraph H is in $\mathrm{L}^{\mathrm{LOGCFL}}$, i.e., in functional* LOGCFL.

As for sequential algorithms, a polynomial time algorithm `opt-`k`-decomp` which, for a fixed k, decides whether a hypergraph has k-bounded hypertree-width and, in this case, computes an optimal, i.e., smallest width hypertree decomposition in normal form is described in [14]. The `opt-`k`-decomp` algorithm is obtained by "uprolling" k`-decomp` in a sequential bottom-up fashion using polynomial space for storing intermediate results while pruning non-optimal partial decompositions. As for many other decomposition methods, the running time of this algorithm to find the hypergraph decomposition is exponential in the parameter k. More precisely, `opt-`k`-decomp` runs in $O(m^{2k}v^2)$ time, where m and v are the number of edges and the number of vertices of H, respectively.

In the next section we will show that the constant k in the exponent of the runtime for computing a hypertree decomposition can most likely not be eliminated.

5 Complexity of Hypertree-Width Computation

In this section we show that determining whether $\mathrm{hw}(H) \leq k$ is NP-complete and actually *fixed-parameter intractable (FP-intractable)* with respect to the parameter k. It follows that, unless some unexpected collapse of FP classes occurs, we cannot eliminate the parameter k from the exponent of the runtime of any algorithm deciding whether a hypergraph H has hypertree-width k, or computing (if possible) a hypertree decomposition of width k of H.

The theory of fixed-parameter tractability or intractability is extensively described in [8]. A problem $\mathcal{P}$ is *fixed-parameter tractable (FP-tractable) w.r.t. parameter k* if there exists a function f and a constant c such that $\mathcal{P}$ can be solved in time $O(f(k)n^c)$, where n is the input size and where $f(k)$ depends only on k and c is a fixed constant independent of k. To prove that a problem is *not* fixed-parameter tractable (*FP-intractable*) one usually reduces another problem, known to be FP-intractable, to it via a *parametric reduction* (see [8]). Such a reduction involves a standard polynomial time reduction f between problem instances, and a mapping g between the parameters.

There is a hierarchy W[1], W[2], W[3], ..., the so called *W-hierarchy*, of classes of parameterized problems that are conjectured to be FP-intractable. A well-known FP-intractable problem at the second level W[2] of this hierarchy is the SET COVER

problem. An instance of SET COVER consists of a hypergraph $H = (V, E)$ and an integer $k \leq |E|$. The problem is to decide whether there exists a set $K \subseteq E$ of k hyperedges covering $V(H)$, i.e., such that $\bigcup_{e \in K} e = V(H)$. The parameter here is k. By FP-reducing SET COVER to the problem of checking whether $\mathrm{hw}(H) \leq k$, we can prove that the latter is W[2]-hard as well. Given that SET COVER (for non-constant parameter k) is NP-hard, the same transformation gives us as a side result that checking whether $\mathrm{hw}(H) \leq k$ is NP-hard in case k is not constant.

Theorem 5. *The problem of deciding whether for a hypergraph H, $\mathrm{hw}(H) \leq k$ is NP-complete and W[2]-hard wrt. parameter k. The same complexity results hold for determining whether $\mathrm{ghw}(H) \leq k$.*

Proof. We state the proof for hypertree-width (hw). First note that the problem is obviously in NP. To show that it is NP-complete and $W[2]$-hard, it suffices to FP-reduce SET COVER to it. Consider an instance I of SET COVER given by a hypergraph $H = (V, E)$ and an integer $k \leq |E|$. Let us define a new hypergraph $H' = (V', E')$ as follows: $V' = V \times \{1, \ldots, 2k+1\}$,

$$E' = \big\{\{(v,i),(w,j)\} \;\big|\; (v,i),(w,j) \in V'\big\} \cup \big\{\, e \times \{1,2,\ldots,2k+1\} \;\big|\; e \in E\big\}.$$

We claim that H has a set cover of size $\leq k$ iff H' has hypertree-width $\leq k$.

The "only if" part is almost trivial to see. Indeed, if there exists a set cover K of size k of H, then a hypertree decomposition of width k of H' is constituted by a tree T consisting of a single node t such that $\chi(t) = V'$ and $\lambda(t) = \{e \times \{1, 2, \ldots, 2k+1\} \mid e \in K\}$.

To see the "if" part of the claim, assume there exists a hypertree decomposition (T, χ, λ) of width k of H'. Then, by construction of H', there must exist a decomposition vertex t of T such that $\chi(t) = V'$. In fact, H' contains as subhypergraph the clique obtained by pairwise relating all vertices of V', and thus any tree decomposition of E' must contain a block containing all vertices of V'. Let

$$S = \{e \in E \mid e \times \{1, 2, \ldots, 2k+1\} \in \lambda(t)\}.$$

Then $|S| \leq |\lambda(t)| \leq k$. We will next show that for each $v \in V$ there exists some $e \in S$ such that $v \in e$, thus S is a set cover of size $\leq k$ of H.

Assume thus that there exists a $v \in V$ such that there is no $e \in S$ for which $v \in e$. Then the elements $(v,1), (v,2), \ldots, (v, 2k+1)$ of $V' = \chi(t)$ must be covered by edges in $\lambda(t)$ of the form $\{v', w'\}$ where $v', w' \in V'$. But for covering $2k+1$ elements by such pairs, at least $k+1$ such pairs would be necessary, which contradicts our assumption that $|\lambda(t)| \leq k$.

The reduction from H to H' is computable in time $O(k \cdot |H|)$ and is thus an FP-reduction.

The same arguments apply if we use the notion of generalized hypertree-width (ghw) instead of hypertree-width (hw). In fact, we have nowhere in this proof made use of the special condition which distinguishes hw from ghw. □

6 Comparing Hypertree-Width to other Hypergraph Invariants

A hypergraph invariant f is *(at least) as good as* invariant g, if there exists a constant c such that whenever for a hypergraph H, $g(H) = k$, then $f(H) \leq c \cdot k$. We say that f *strongly dominates* g if f is at least as good as g and there is a class $\mathcal{H}$ of hypergraphs for which f is bounded (i.e., $\exists k \, \forall H \in \mathcal{H} : f(H) \leq k$), but g is unbounded. We say that two invariants f and g are *equivalent* if each is as good as the other one.

We start by discussing some hypergraph invariants that are generalizations of sophisticated graph invariants, and then report some results on comparing invariants used in Constraint Satisfaction, Artificial Intelligence and Database Theory to hypertree-width.

Hyperlinkedness. Let H be a hypergraph, $M \subseteq E(H)$ and $C \subseteq V(H)$. C is *M-big*, if it intersects more than half of the edges of M, that is, $|\{e \in M \mid e \cap C \neq \emptyset\}| > \frac{|M|}{2}$. Note that if $S \subseteq E(H)$, then $H \setminus \bigcup S$ has at most one M-big connected component. Let $k \geq 0$ be an integer. A set $M \subseteq E(H)$ is *k-hyperlinked*, if for any set $S \subseteq E(H)$ with $|S| < k$, $H \setminus \bigcup S$ has an M-big component. The largest k for which H contains a k-hyperlinked set is called *hyperlinkedness of H*, $\mathrm{hlink}(H)$. Hyperlinkedness is an adaptation of the linkedness of a graph [25] to the setting of hypergraphs.

Brambles. Let H be a hypergraph. Sets $X_1, X_2 \subseteq V(H)$ *touch* if $X_1 \cap X_2 \neq \emptyset$ or there exists an $e \in E(H)$ such that $e \cap X_1 \neq \emptyset$ and $e \cap X_2 \neq \emptyset$. A *bramble of H* is a set B of pairwise touching connected subsets of $V(H)$. This is defined in analogy to brambles of graphs [25]. The *hyper-order of a bramble B* is the least integer k such that there exists a set $R \subseteq E(H)$ with $|R| = k$ and $\bigcup R \cap X \neq \emptyset$ for all $X \in B$. The *hyperbramble number* $\mathrm{hbramble\text{-}no}(H)$ of H is the maximum of the hyper-orders of all brambles of H.

Theorem 6 ([2]). *For each hypergraph H,* $\mathrm{hlink}(H) \leq \mathrm{hbramble\text{-}no}(H) \leq \mathrm{mw}(H) \leq \mathrm{ghw}(H) \leq \mathrm{mon\text{-}mw}(H) = \mathrm{hw}(H) \leq 3 \cdot \mathrm{hlink}(H) + 1$.

Corollary 61 *The hypergraph invariants* hlink, $\mathrm{hbramble\text{-}no}$, mw, ghw, $\mathrm{mon\text{-}mw}$, *and* hw *are all equivalent.*

Of particular interest is the result that the generalized hypertree-width $\mathrm{ghw}(H)$ of a hypergraph H is at most a factor 3 smaller than the hypertree-width $\mathrm{hw}(H)$. This is important, because while it is currently an open problem whether $\mathrm{ghw}(H) \leq k$ is decidable in polynomial time for constants k, the notion of generalized hypertree-width is by many considered the best possible measure of cyclicity of a hypergraph. For example, Cohen, Jeavons, and Gyssens [4] recently introduced a general framework for hypergraph decomposition in the context of which they introduced the concept of an *acyclic guarded cover* as their most general considered decomposition guaranteeing tractability of the underlying problems (i.e., satisfying the above Condition 3). It turns out, however, that an acyclic guarded cover can be equivalently defined as the set of nodes of a generalized hypertree decomposition, and that the corresponding notion of width *precisely* coincides with the notion of generalized hypertree-width. This provides further evidence of the naturalness and importance of this notion.

The following hypergraph invariants were considered in AI, and, in particular, in the area of constraint processing.

Biconnected Components (short: BICOMP) [9]. Any graph $G = (V, E)$ can be decomposed into a pair $\langle T, \chi \rangle$, where T is a tree, and the labeling function χ associates to each vertex of T a biconnected component of G. The *biconnected width* of a hypergraph H, denoted by BICOMP-width(H), is the maximum number of vertices over the biconnected components of the primal graph of H.

Cycle Cutset and Hypercutset (short: CUTSET) [5]. A *cycle cutset* of a hypergraph H is a set $S \subseteq V(H)$ such that the subhypergraph of H induced by $V(H) - S$ is acyclic. The CUTSET-width of H is the minimum cardinality over all its possible cycle cutsets. A generalization of this is the method of hypercutsets, short HYPERCUTSET (for a definition, see [15]).

Tree Clustering (short: TCLUSTER) [7]. The *tree clustering* method is based on a triangulation algorithm which transforms the primal graph $G = (V, E)$ of any hypergraph H into a chordal graph G'. The maximal cliques of G' are then used to build the hyperedges of an acyclic hypergraph H'. The *tree-clustering width* (short: TCLUSTER *width*) of H is 1 if H is an acyclic hypergraph; otherwise it is equal to the maximum cardinality over the cliques of the chordal graph G'.

The Hinge Method (HINGE) [19,18]. This is an interesting decomposition method generalizing acyclic hypergraphs. For space reasons, we omit a formal definition. Computing the HINGE-width of a hypergraph is feasible in polynomial time [19,18]. One can also combine the methods HINGE and TCLUSTER, yielding the more general method $\text{HINGE}^{\text{TCLUSTER}}$.

Theorem 7 ([15]). *Hypertree-width strongly dominates treewidth,* BICOMP*-width,* CUTSET*-width,* HYPERCUTSET*-width,* TCLUSTER*-width,* HINGE*-width, and* $\text{HINGE}^{\text{TCLUSTER}}$*-width.*

7 Heuristics for Hypertree Decomposition

Recall that the algorithm opt-k-decomp decides, for a fixed k, whether a given hypergraph has k-bounded hypertree-width and, if so, computes a hypertree decomposition of minimal width. Although opt-k-decomp runs in polynomial time, it is too slow and needs a huge amount of space when applied to large hypergraphs. Therefore, recent research focuses on heuristic approaches for the construction of hypertree decompositions. Of particular interest is the application of well-known heuristics from other areas to hypertree decomposition.

Recall that a hypertree decomposition is in principle the same as a tree decomposition satisfying two additional conditions. The first one leads from a tree decomposition to a generalized hypertree decomposition and says that for every $t \in V(T)$ it holds that $\chi(t) \subseteq \bigcup \lambda(t)$, and the second one is the *special condition* leading from a generalized hypertree decomposition to a hypertree decomposition. Note that the *special condition* was introduced in order to be able to prove the polynomial runtime of opt-k-decomp. Hence, the *special condition* can be ignored when considering heuristic algorithms, and thus, one actually aims at computing *generalized* hypertree decompositions by using heuristics.

So, when constructing hypertree decompositions via tree decomposition heuristics, there is only one additional condition we have to satisfy. This condition forces the λ-labels to cover the χ-labels. A natural approach to obtain a hypertree decomposition

from a tree decomposition is therefore to implement this condition in a straight-forward way by set covering, i.e., to use set covering algorithms in order to compute the λ-labels of the hypertree decomposition based on the χ-labels given by the tree decomposition. In this way, it is possible to use tree decomposition heuristics (together with set covering heuristics) for the heuristic construction of hypertree decompositions.

This approach was firstly applied by McMahan [23] who obtained surprisingly good results within a small amount of time. McMahan used *Bucket Elimination* [6] in combination with several variable ordering heuristics. Obviously, to construct hypertree decompositions in this way, any underlying tree decomposition method can be used. Moreover, also branch decomposition heuristics are applicable [28], since every branch decomposition of width k can be transformed into a tree decomposition of width at most $3k/2$ [26].

Another approach for heuristic hypertree decomposition is dual to the above ones in the sense that we obtain a hypertree where the λ-labels are given and appropriate χ-labels have to be set. This can be easily achieved by building a tree decomposition of the dual graph. The *dual graph* of a hypergraph is obtained by creating a vertex for each hyperedge and connecting two vertices if the corresponding hyperedges have a common vertex. This dual graph, however, has too many edges for our purposes, i.e., the resulting hypertree-width would be higher than necessary. Moreover, a hypertree decomposition resulting from this procedure is always a query decomposition [13] whose width is always larger than or equal to the hypertree-width of a hypergraph. However, by using pre- and post-processing heuristics, it is possible to overcome both problems.

Finally, let us mention a further heuristic approach. It is based on hypergraph clustering resp. hypergraph partitioning. There exist several heuristics in the literature for building clusters of strongly connected hyperedges in a hypergraph such that there are as less hyperedges as possible between the clusters. By using such methods, it is possible to construct a hypertree decomposition in such a way that the clusters are recursively partitioned and in each step a *special hyperedge* is added [21]. During this process, for each cluster a hypertree-node is created whose λ-labels are exactly the hyperedges separating the subclusters of the current cluster. Afterwards, it is possible to connect these hypertree-nodes in such a way that the resulting hypertree is indeed a hypertree decomposition of the hypergraph (cf. [21]).

8 Applications

There are many problems in various domains of Computer Science whose underlying structure is best described as a hypergraph and that are efficiently solvable if this structure is acyclic. We next show that, for most of them, the notion of (generalized) hypertree-width provides a technique for solving efficiently large classes of instances that were believed to be intractable, according to previous known methods.

A very important example of such problems is the NP-hard *Constraint Satisfaction Problem (CSP)*, which is an important goal of AI research. Constraint satisfaction is a central issue of *problem solving* and has an impressive spectrum of applications [24]. A constraint (S_i, R_i) consists of a *constraint scope* S_i, i.e., a list of variables, and an associated *constraint relation* r_i containing the legal combinations of values. A CSP

consists of a set $\{(S_1, r_1), (S_2, r_2), \ldots, (S_q, r_q)\}$ of constraints whose variables may overlap. A solution to a CSP consists of an assignment of values to all variables such that all constraints are simultaneously satisfied. By *solving* a CSP we mean determining whether the problem has a solution at all (i.e., checking for *constraint satisfiability*), and, if so, compute one solution.

Example 3. Consider the CSP I^a consisting of constraints $\{C_1^a, \ldots, C_9^a\}$ where, for each constraint C_i^a, the constraint relation r_i^a encodes some required property for the variables occurring together in the corresponding scope S_i^a, and the constraint scopes are the following: $S_1^a(3, 4, 5, 6, 7, 8)$; $S_2^a(12, 16, 17, 18, 19)$; $S_3^a(7, 9, 10)$; $S_4^a(1, 11, 17, 19)$; $S_5^a(1, 2, 3, 4, 5, 6)$; $S_6^a(5, 6, 7, 8, 9)$; $S_7^a(12, 15, 16, 18, 19)$; $S_8^a(12, 13, 14, 15, 18, 19)$; $S_9^a(11, 12, 17, 18, 19)$.

The constraint hypergraph of a CSP I is the hypergraph $H(I)$ whose vertices are the variables of the CSP and whose hyperedges are the sets of all those variables which occur together in a constraint scope. It is well known that CSPs with *acyclic* constraint hypergraphs are polynomially solvable [5]. For instance, our example CSP instance I^a is acyclic, as its hypergraph has a join tree. In fact, it is easy to check that the tree shown in Figure 1 (on the right) is a join tree of hypergraph $H(I^a)$. Intuitively, the efficient behavior of acyclic instances is due to the fact that they can be evaluated by processing any of their join trees bottom-up by performing upward semijoins (in database lingo) [30]. That is, starting from the leaves, for each vertex v of the tree, we may filter out of its parent $p(v)$ the tuples of values from $p(v)$'s constraint relation that do not match with any tuple in the relation of v. At the end, if the relation in the root is not empty, we know that the given instance has a solution. This procedure takes $O(nm \log m)$ time, where m is the size of the largest relation and n is the number of constraints. Note that we do not distinguish here among join tree vertices and constraints, because join tree vertices correspond to hyperedges and hence to constraints (assuming, w.l.o.g., that there is no pair of constraints with exactly the same scopes). Recall that in general even computing small outputs, e.g. just one solution, requires exponential time (unless P $=$ NP) [3], indeed the typical worst case cost for CSP algorithms is $O(m^{n-1} \log m)$.

The idea behind CSP algorithms based on generalized hypertree decompositions is to transform a CSP I into an equivalent acyclic CSP I', by organizing its scopes into a polynomial number of clusters that may suitably be arranged as a tree. Consider a generalized hypertree decomposition of $H(I)$ and some vertex v of this decomposition. We can combine the constraints in $\lambda(v)$ in a unique constraint over the only variables listed in $\chi(v)$. Building this fresh constraint takes $O(m^{|\lambda(v)|-1} \log m)$ time. It is easy to see that, after this phase, we get a new CSP instance I', which is acyclic and solution equivalent to the original instance I. Therefore, we can eventually solve this instance in time $O(n' m^{w-1} \log m)$, where w is the decomposition-width and n' is the number of vertices in the decomposition tree, which is bounded by the number of hypergraph vertices (CSP variables). Note that, for classes of CSPs having small (bounded) width, solving these problems by exploiting hypertree decompositions may lead to a tremendous speed-up. Indeed, hypertrees with the smallest width say to us precisely the best way of combining together constraints of I, in order to obtain a nice acyclic equivalent instance to be solved efficiently.

Example 4. Consider a CSP instance I^c with the following constraint scopes:

$S_1(1, 2, 3)$; $S_2(1, 4, 5, 6)$; $S_3(3, 4, 7, 8)$; $S_4(5, 7)$; $S_5(6, 8, 9)$; $S_6(7, 9, 10)$; $S_7(5, 9)$; $S_8(1, 11)$; $S_9(11, 12, 18)$; $S_{10}(12, 13, 19)$; $S_{11}(13, 14)$; $S_{12}(14, 15, 18)$; $S_{13}(15, 16, 19)$; $S_{14}(16, 17, 18)$; $S_{15}(1, 17, 19)$;

The associated hypergraph $H(I^c)$ is shown in Figure 1. The generalized hypertree-width of this hypergraph is 2 and a decomposition having this (optimal) width is shown in Figure 1, on the left. Following the "instructions" encoded in this decomposition, we build exactly the acyclic instance I^a in Example 3. Then, by exploiting hypertrees, we know that I^c may be solved in $O(9m \log m)$ time, in the worst case, which is clearly quite good, if compared with the traditional worst case $O(m^{14} \log m)$.

Though we focused on constraint satisfiability, all the above considerations immediately apply to a large number of important problems that, as CSP, are efficiently solvable if their hypergraph structure is acyclic. We just mention here a few examples, such as the game theory problem of computing pure Nash equilibria in graphical games [10], and various database problems, e.g., the problem of conjunctive query containment [20], or the problem of evaluating *Boolean conjunctive queries* over a relational database [22] (for a discussion of this and other equivalent problems, see [11]).

9 Open Problems and Future Research

We believe that hypertree decompositions and hypertree-width are interesting concepts deserving further investigations. The following problems are of particular interest: (1) Is it possible to check whether $\mathrm{ghw}(H) \leq k$ in polynomial time for each constant k? (2) Are there other hypergraph invariants (and associated decompositions) that fulfill the three criteria given in Section 1 and that strongly generalize hypertree-width? (3) Can we find a deterministic algorithm for computing a k-width hypertree decomposition whose worst case runtime is significantly better than n^{2k}? (4) Is it possible to find some heuristic method for computing "good" hypertree decompositions for an overwhelmingly large number of realistic examples stemming from various applications?

References

1. I. Adler. Marshals, monotone marshals, and hypertree width. *Journal of Graph Theory* 47, pages 275–296, 2004.
2. I. Adler, G. Gottlob, and M. Grohe. Hypertree-width and related hypergraph invariants. Manuscript, submitted for publication, available from the authors.
3. A. K. Chandra and P. M. Merlin. Optimal implementation of conjunctive queries in relational databases. In *Proc. STOC'77*, pages 77–90, 1977.
4. D. A. Cohen, P. G. Jeavons, and M. Gyssens. A unified theory of structural tractability for constraint satisfaction and spread cut decomposition. In *Proc. IJCAI'05*, pages 72–77, 2005.
5. R. Dechter. Constraint networks. In *Encyclopedia of Artifi cial Intelligence*, second edition, Wiley & Sons, pages 276-–285, 1992.
6. R. Dechter and J. Pearl. Network-based heuristics for constraint satisfaction problems. *Artifi cial Intelligence* 34(1), pages 1–38, 1987.
7. R. Dechter and J. Pearl. Tree clustering for constraint networks. *Artifi cial Intelligence* 38(3), pages 353–366, 1989.

8. R. G. Downey and M. R. Fellows. *Parameterized Complexity*. Springer, 1999.
9. E. C. Freuder. A sufficient condition for backtrack–bounded search. *Journal of the ACM* 32(4), pages 755–761, 1985.
10. G. Gottlob, G. Greco, and F. Scarcello. Pure Nash equilibria: Hard and easy games. *Journal of Artificial Intelligence Research (JAIR)*, 2005. To appear. Preliminary version in: *Proc. TARK'03*, 2003.
11. G. Gottlob, N. Leone, and F. Scarcello. The complexity of acyclic conjunctive queries. *Journal of the ACM* 48(3), pages 431–498, 2001. Preliminary version in: *Proc. FOCS'98*, 1998.
12. G. Gottlob, N. Leone, and F. Scarcello. Computing LOGCFL certificates. *Theoretical Computer Science* 270(1-2), pages 761–777, 2002. Preliminary version in: *Proc. ICALP'99*, 1999.
13. G. Gottlob, N. Leone, and F. Scarcello. Hypertree decompositions and tractable queries. *Journal of Computer and System Sciences (JCSS)* 64(3), pages 579–627, 2002. Preliminary version in: *Proc. PODS'99*, 1999.
14. G. Gottlob, N. Leone, and F. Scarcello. On tractable queries and constraints. In *Proc. DEXA'99*, pages 1–15, 1999.
15. G. Gottlob, N. Leone, and F. Scarcello. A comparison of structural CSP decomposition methods. *Artificial Intelligence* 124(2), pages 243–282, 2000. Preliminary version in: *Proc. IJCAI'99*, 1999.
16. G. Gottlob, N. Leone, and F. Scarcello. Robbers, marshals, and guards: Game-theoretic and logical characterizations of hypertree width. In *Proc. PODS'01*, pages 195–206, 2001.
17. G. Gottlob and R. Pichler. Hypergraphs in model checking: Acyclicity and hypertree-width versus clique-width. *Siam Journal of Computing* 33(2), pages 351–378, 2004.
18. M. Gyssens, P. G. Jeavons, and D. A. Cohen. Decomposing constraint satisfaction problems using database techniques. *Artificial Intelligence* 66, pages 57–89, 1994.
19. M. Gyssens, and J. Paredaens. A decomposition methodology for cyclic databases. In *Advances in Database Theory*, vol. 2, pages 85–122, 1984.
20. Ph. G. Kolaitis and M. Y. Vardi. Conjunctive-query containment and constraint satisfaction. *Journal of Computer and System Sciences (JCSS)* 61, pages 302–332, 2000.
21. T. Korimort. Constraint satisfaction problems – Heuristic decomposition. PhD thesis, Vienna University of Technology, April 2003.
22. D. Maier. The theory of relational databases. Computer Science Press, 1986.
23. B. McMahan. Bucket eliminiation and hypertree decompositions. Implementation report, Institute of Information Systems (DBAI), TU Vienna, 2004.
24. J. Pearson and P. G. Jeavons. A survey of tractable constraint satisfaction problems. Technical report CSD-TR-97-15, Royal Halloway University of London, 1997.
25. B. Reed. Tree width and tangles: A new connectivity measure and some applications. In *Surveys in Combinatorics*, volume 241 of LNS, pages 87–162. Cambridge University Press, 1997.
26. N. Robertson and P. D. Seymour. Graph minors. X. Obstructions to tree-decomposition. *Journal of Combinatorial Theory, Series B* 52, pages 153–190, 1991.
27. W. L. Ruzzo. Tree-size bounded alternation. *Journal of Computer and System Sciences (JCSS)* 21(2), pages 218–235, 1980.
28. M. Samer. Hypertree-decomposition via branch-decomposition. In *Proc. IJCAI'05*, pages 1535–1536, 2005.
29. P. D. Seymour and R. Thomas. Graph searching and a min-max theorem for tree-width. *Journal of Combinatorial Theory, Series B* 58, pages 22–33, 1993.
30. M. Yannakakis. Algorithms for acyclic database schemes. In *Proc. VLDB'81*, pages 82–94, 1981.