Computing optimal hypertree decompositions with SAT

André Schidler *, Stefan Szeider

Algorithms and Complexity Group, TU Wien, Vienna, Austria

Abstract

Hypertree width is a prominent hypergraph invariant with many algorithmic applications in constraint satisfaction and databases. We propose two novel characterisations for hypertree width in terms of linear orderings. We utilize these characterisations to obtain SAT, MaxSAT, and SMT encodings for computing the hypertree width exactly. We evaluate the encodings on an extensive set of benchmark instances and compare them to state-of-the-art exact methods for computing optimal hypertree width. Our results show that our approach outperforms these state-of-the-art algorithms.

© 2023 The Authors. Published by Elsevier B.V. This is an open access article under the CC BY license (http://creativecommons.org/licenses/by/4.0/).

1. Introduction

Hypertree width is a popular hypergraph invariant which was introduced by Gottlob et al. [19]. Hypertree width is of fundamental nature, as underpinned by the existence of combinatorial, game-theoretic, as well as logical characterisations [20]. In their original paper, Gottlob et al. [19] showed that critical NP-hard problems arising in databases and constraint satisfaction are polynomial-time tractable for instances whose associated hypergraph has bounded hypertree width. The hypergraph invariant has found many further applications, including Projected Solution Counting, Solution Enumeration (with polynomial delay), Constraint Optimization, and Combinatorial Auctions (see, for example, the survey article [22]).

Key to these tractability results is the fact that for any fixed constant bound \( W \), one can decide in polynomial time whether a given hypergraph has hypertree width up to \( W \), and in the positive case compute a witnessing hypertree decomposition of width \( W \). Thus, in terms of parameterized complexity \([11]\), the problem of recognizing hypergraphs of hypertree width \( W \) is in XP, when parameterized by \( W \). However, the problem is known to be \( W[2]-hard \) [21] and hence unlikely to be fixed-parameter tractable. The original XP-algorithm by Gottlob et al. [19] was later improved by Gottlob and Samer [18], and their implementation of the algorithm (det-k-decomp) represented for several years the state of the art for practically computing optimal hypertree decompositions.

We propose a new practical approach for computing the exact hypertree width of hypergraphs. We follow a logical approach which was initiated by Samer and Veith [32] for tree decompositions and was later successfully used for other (hyper)graph width measures, including clique-width [25], treecut width and treedepth [17], fractional hypertree width [13], and twin-width [35]. For hypertree width, we had to introduce several new concepts and ideas to make this approach work.
The general idea is to use a polynomial-time encoding algorithm, which takes as input a hypergraph $H$ and an integer $W$, and produces a propositional formula $F(H, W)$, such that $F(H, W)$ is satisfiable if and only if the hypertree width of $H$ is at most $W$. By trying systematically different values of $W$, we can determine the smallest $W$ for which $F(H, W)$ is satisfiable, i.e., the hypertree width of $H$. Subsequently, we use a polynomial-time decoding algorithm which translates a satisfying assignment of $F(H, W)$ into a hypertree decomposition of $H$ of width $W$.

We propose two encodings, based on two different characterisations of hypertree width. Both characterisations are ordering-based, where we arrange the vertices of the given hypergraph in a linear ordering subject to certain constraints. Successful ordering-based encodings for treewidth and fractional hypertree width \cite{population_1, population_2} already used ordering-based characterisations of the corresponding width measures (for treewidth, this characterisation uses the well-known characterisation of graphs of bounded tree-width in terms of partial $k$-trees \cite{61}). However, for hypertree width, an ordering based characterisation is not straightforward. What makes it challenging to express hypertree width in terms of a linear ordering is the Special Condition in the definition of hypertree decompositions (see Section 2), which is formulated in terms of the descendency relation in the decomposition tree. However, we succeeded in formulating two characterisations in such a way that we could base compact and efficient SAT encodings on it.

Our first encoding, based on the augmented hypertree ordering characterisation of hypertree width, uses in addition to the linear ordering additional relational information, which allows us to express the Special Condition in a way that closely relates to the original definition of hypertree width.

Our second encoding, based on the pure hypertree ordering characterisation of hypertree width, avoids the additional relational information and is therefore more compact. This characterisation is conceptually more elegant than the augmented one, but requires more formal arguments to establish its equivalence with the original definition of hypertree width.

For bounding the width of an augmented or pure hypertree ordering, we need to compute small hyperedge covers of vertex sets. For this purpose, we use in our encodings not only propositional cardinality constraints \cite{sat} (as Samer and Veith did), but also soft clauses in conjunction with a MaxSAT solver and certain arithmetic constraints. During the solving process, these arithmetic constraints are mapped to propositional logic in an incremental fashion. This incremental encoding is handled by an SMT (SAT Modulo Theory) solver \cite{g15, g16}, where a First-Order Logic solver (handling the arithmetic constraints) interacts with the SAT solver.

We implemented the encodings based on augmented and pure hypertree orderings and tested them on an extensive set of benchmark instances (Hyperbench), consisting of real-world hypergraphs from various application domains, with a number of vertices and hyperedges ranging up to 2900. We tested two variants, one where the cardinality constraints are encoded using binary counters (plain SAT and MaxSAT) and one where the cardinality constraints are encoded by arithmetic constraints that are dealt with by the theory component of a SAT Modulo Theory solver (SMT). We compared the two encodings with each other, and also with the newest version of Gottlob and Samer’s combinatorial XP-algorithm new-det-k-decomp \cite{g17, g18}. The results are highly encouraging and show that each of our methods improve upon the state of the art by solving up to 621, or almost 50%, more instances. Furthermore, our new methods work complementary and can solve even more instances in a portfolio approach, combining the different encodings and solver paradigms.

This paper is based on and extends results presented in preliminary form in an ALENEX/20 paper \cite{g19}, where a predecessor of the augmented hypertree ordering was introduced, and an IJCAI’21 paper \cite{g20}, where a predecessor of the pure hypertree ordering was introduced. We note that computing hypertree width was subject of the 4th edition of the Parameterized Algorithms and Computational Experiments Challenge (PACE 2019 \cite{g21}), where a variant of the encoding based on augmented hypertree ordering won the exact track, tailed by a new implementation of Gottlob and Samer’s algorithm. In addition to full proofs, improvements, and additional experimental results, this paper provides a comprehensive exposition of both hypertree ordering characterisations of hypertree width and their performance on different instances. Furthermore, our results show that our encodings and different solver paradigms work in a complementary fashion and that therefore even better results can be achieved by using both our encodings together with a MaxSAT and SMT solver.

The remainder of the paper is organized as follows. In Section 2 we give basic definitions on hypergraphs, edge covers, and hypertree decompositions. Afterwards in Section 3, we introduce our new ordering-based characterisation of hypertree width, whose correctness we establish in Appendix A and Appendix B. In Section 4, we explain how the new characterisations can be encoded as a SAT problem. Finally in Section 5, we present our experimental results.

2. Preliminaries

A hypergraph $H$ consists of a set $V(H)$ of vertices and a set $E(H)$ of hyperedges, each hyperedge is a subset of $V(H)$. A hypergraph $H'$ is a partial hypergraph of hypergraph $H$ if $E(H') \subseteq E(H)$ and $V(H') = \bigcup_{e \in E(H')} e$.

The primal graph (or 2-section) of a hypergraph $H$ is the graph $G(H)$ with vertex set $V(H)$ and edge set $E(G(H)) = \{ \{u, v\} \mid u \neq v \text{ and there is some } e \in E \text{ such that } \{u, v\} \subseteq e\}$. A hypergraph is connected if its primal graph is connected, otherwise it is disconnected. A connected component of a hypergraph $H$ is a maximal connected partial hypergraph of $H$. A hypergraph $H$ decomposes into its connected components $H_1, \ldots, H_s$, where $V(H_i) \cap V(H_j) = \emptyset$ and $E(H_i) \cap E(H_j) = \emptyset$, $1 \leq i < j \leq s$. We write $CC(H) = \{H_1, \ldots, H_s\}$ for the set of connected components of $H$.

Consider a hypergraph $H$ and a set $S \subseteq V$. An edge cover of $S$ (with respect to $H$) is a set $F \subseteq E(H)$ such that for every $v \in S$ there is some $e \in F$ with $v \in e$. The size of an edge cover is its cardinality. Given an edge cover $F$, we use the shorthand $\bigcup F := \bigcup_{e \in F} e$. 

A. Schidler and S. Szeider

Artificial Intelligence 325 (2023) 104015
A tree decomposition of a hypergraph \( H = (V, E) \) is a pair \( (T, \chi) \) where \( T \) is a tree with vertex set \( V(T) \) and edge set \( E(T) \) and \( \chi \) is a mapping that assigns to each \( t \in V(T) \) a set \( \chi(t) \subseteq V(H) \), called the bag at \( t \), such that the following properties hold:

**T1** for each \( v \in V(H) \) there is some \( t \in V(T) \) with \( v \in \chi(t) \) ("\( v \) is covered by \( t \)").

**T2** for each \( e \in E(H) \) there is some \( t \in V(T) \) with \( e \subseteq \chi(t) \) ("\( e \) is covered by \( t \)").

**T3** for any three \( t, t', t'' \in V(T) \) where \( t' \) lies on the path between \( t \) and \( t'' \) in \( T \), we have \( \chi(t) \cap \chi(t'') \subseteq \chi(t') \) ("bags containing the same vertex are connected").

We assume the tree \( T \) to be rooted at some arbitrary node \( r \in V(T) \), as this will be needed for extending tree decompositions to hypertree decompositions below.

We say that a vertex \( v \in V(H) \) is forgotten at node \( t \) if \( v \in \chi(D) \) but \( v \) is not in the bag of \( t \)'s parent. Every vertex in the bag of the root node \( r \), is forgotten at \( r \). We observe that each vertex \( v \) is forgotten at exactly one node \( t \) due to T3, hence we can write \( f(v) = t \).

A hypertree decomposition [19] of \( H \) is a triple \( D = (T_D, \chi_D, \lambda_D) \) where \( (T_D, \chi_D) \) is a tree decomposition of \( H \), and \( \lambda_D \) is a mapping that assigns to each \( t \in V(T_D) \) an edge cover \( \lambda_D(t) \subseteq E(H) \) of \( \chi_D(t) \). The width of \( D \) is the size of the largest edge cover in \( \lambda_D \). Moreover, the rooted tree \( T_D \) satisfies in addition to T1–T3 also a certain Special Condition (T4). To formulate the Special Condition, we call a vertex \( v \) to be omitted at a node \( t \in V(T_D) \), if \( v \notin \chi_D(t) \), but \( \lambda_D(t) \) contains a hyperedge \( e \) with \( v \in e \). The Special Condition now states the following:

**T4** If a vertex \( v \) is omitted at \( t \), then it must not appear in the bag \( \chi_D(t') \) of any descendant node \( t' \) of \( t \).

In other words, T4 states that if \( t, t' \in V(T_D) \) are nodes such that \( t' \) is a descendant of \( t \), then for each \( e \in \lambda_D(t) \) we have \( (e \setminus \chi_D(t')) \cap \chi_D(t') = \emptyset \). The hypertree width \( \text{htw}(H) \) of \( H \) is the smallest width over all hypertree decompositions of \( H \). We say a hypertree decomposition \( D \) of a hypergraph \( H \) is optimal if the width of \( D \) equals \( \text{htw}(H) \). See Fig. 1 for an example.

If the decomposition is not required to satisfy the Special Condition, then we call it a generalized hypertree decomposition, and we define accordingly the generalized hypertree width \( \text{ghtm}(H) \) of \( H \) as the smallest width over all generalized hypertree decompositions of \( H \). Clearly \( \text{ghtm}(H) \leq \text{htw}(H) \). It is already NP-hard to decide whether a given hypergraph has generalized hypertree width \( \leq 2 \) [15], hence dropping the Special Condition increases the parameterized complexity of the recognition problem from XP to para-NP. The even more general parameter fractional hypertree width (with the same para-NP-hard recognition problem [15]) arises when one considers fractional edge covers of the bags instead of edge covers [24]. Powerful decomposers have been developed for generalized and fractional hypertree width [13,27].

To avoid trivial cases, we consider only hypergraphs \( H \) where each \( v \in V(H) \) is contained in at least one \( e \in E(H) \). Consequently, every considered hypergraph \( H \) has an edge cover, and \( \text{htw}(H) \) is always defined. If \( |V(H)| = 1 \) then \( \text{htw}(H) = \text{ghtm}(H) = 1 \).

We will also focus on connected hypergraphs, since we can proceed component-wise to compute an optimal hypertree decomposition:

**Proposition 1.** For every hypergraph \( H \) we have \( \text{htw}(H) = \max_{H' \in \text{CC}(H)} \text{htw}(H') \).

### 3. Elimination orderings for hypertree width

For this section, we consider a fixed, connected hypergraph \( H \), and a linear ordering \( v_1 < \cdots < v_n \) of \( V(H) \). We think of the ordering as an elimination ordering in the very same sense as has been used in the context of tree decompositions for graphs [8]. We eliminate one vertex after the other from the primal graph. Each time a vertex is eliminated, we make all its (remaining) neighbors adjacent (see, for example, [8]), where newly added edges are referred to as fill-in edges. For our
purposes and the forthcoming encodings, it is convenient to consider this elimination process on a directed version of the primal graph, where edges are oriented with respect to the linear ordering.

To that effect, let the set of active arcs \( A \) be the smallest subset of \( V(H) \times V(H) \) such that

1. if \( (u, v) \in E(G(H)) \) and \( u \prec v \) then \( (u, v) \in A \), and
2. if \( (u, v) \in A \), \( (u, w) \in A \) and \( v \prec w \) then \( (v, w) \in A \).

We can now define the following central concept: For a fixed linear ordering \( \prec \) of \( V(H) \), a rooted tree \( T_\prec \) is a canonical tree for \( \prec \) if

1. \( V(T_\prec) = \{ \tau_\prec(v) \mid v \in V(H) \} \), i.e., the tree contains one node for each of \( H \)'s vertices,
2. if \( r \) is the \( \prec \)-largest vertex in \( V(H) \), then \( \tau_\prec(r) \) is the root of \( T_\prec \), and
3. for any \( u \in V(H) \setminus \{ r \} \), if \( v \) is the \( \prec \)-smallest vertex with \( (u, w) \in A \), then \( \tau_\prec(w) \) is the parent of \( \tau_\prec(u) \) in \( T_\prec \).

Since all canonical trees are isomorphic, we call \( T_\prec \) the canonical tree for \( \prec \).

In the sequel, we will also use the transitive closure \( A^* \) of \( A \), i.e., the smallest set containing \( A \) with the property that whenever \( (u, v) \in A^* \) and \( (v, w) \in A^* \), then also \( (u, w) \in A^* \). We note that in the last item of the definition of \( T_\prec \) we could have used \( A^* \) instead of \( A \) without changing the definition.

Using these definitions, we can construct a generalized hypertree decomposition \( D = (T_\prec, \chi_D, \lambda_D) \) based on \( \prec \), where for each \( v \in V(H) \):

- \( \chi_D(\tau(v)) := \{ v \} \cup \{ w \mid (v, w) \in A \} \), and
- \( \lambda_D(v) \) is a smallest edge cover of \( \chi_D(\tau(v)) \).

Fig. 2 (right) shows an example of such a decomposition. Here, we can see an inherent problem of this way of characterizing hypertree width in terms of elimination orderings that we need to address (“accumulating hyperedges”): The bag of the root cannot be covered without violating T4, as any vertex adjacent to \( c \) occurs in the bag of some descendant. In general this problem can occur anywhere in the decomposition, not only at the root. In the example we can resolve the problem by adding the white vertices. We observe that even when we ignore the redundant bags above \( \tau_\prec(e) \), we do not get the same decomposition as in Fig. 1, since this decomposition tree is not the canonical tree of any ordering.

We present two characterisations of hypertree width in terms of elimination orderings that address this issue. The pure hypertree ordering uses properties of hypertree decompositions to implicitly constrain the elimination ordering, such that the Special Condition is not violated. The augmented hypertree ordering augments the elimination ordering by an equivalence relation to address the issue of accumulating hyperedges. The two characterisations will provide the basis for SAT/MaxSAT/SMT encodings that we will present in Section 4.

### 3.1. Pure hypertree orderings

Before we can discuss this characterisation, we need further concepts and considerations.

We define for each vertex \( v \in V(H) \) the set

\[
\chi_\prec(v) := \{ v \} \cup \{ w \mid (v, w) \in A \}.
\]

This definition is closely related to the construction for the generalized hypertree decomposition’s bags at the beginning of the section.
For any two vertices \( u, w \in V(H) \), such that \( u < w \), we define their arc-path

\[
P(u, w) := \{ u, w \} \cup \{ v \mid (u, v) \in A^*, (v, w) \in A^* \},
\]

and we say a vertex \( w \in V(H) \) is arc-reachable from \( u \) if \( (u, w) \in A^* \).

In Fig. 2 (left), we can find arc-paths by simply following the arcs from left to right. This illustrates the motivation for arc-paths: whenever one vertex is arc-reachable from another, the corresponding nodes have a descendency relationship in the canonical tree.

We now introduce the construction that ensures that the Special Condition holds. Assuming a mapping \( \lambda_\prec \) that assigns an edge cover for \( \chi_\prec \) to each vertex, we divide the forbidden vertices—vertices that would violate the Special Condition—at any vertex \( w \) into two disjoint sets \( B(w) \) and \( R(w) \).

\[
R(w) := \{ u \mid (v, w) \in A^* \text{ and } \lambda_\prec(v) \subseteq \lambda_\prec(w), \text{ for every } v \in P(u, w) \}
\]

\[
B(w) := \{ u \mid (v, w) \in A^* \text{ and } \lambda_\prec(v) \setminus \lambda_\prec(w) \neq \emptyset, \text{ for some } v \in P(u, w) \}
\]

The T4-violations in \( R(w) \) can be repaired: whenever every vertex on the arc-path between \( u \) and \( w \) has as its edge cover a superset of \( u \)’s edge cover, we can omit \( u \) at \( w \). If this property holds, we can add \( \chi_\prec(u) \) to all the bags along the arc-path. Whenever we can find an ordering and an edge cover such that \( B(w) \) is empty for each vertex \( w \in V(H) \), we can convert the ordering into a hypertree decomposition.

In Fig. 2 (right), \( R \) is represented by the white vertices. For the root, \( B \) consists of all vertices except \( c \) and the white vertices.

These definitions in hand, we can now state our first ordering-based characterisation of hypertree width.

**Definition 2 (Pure hypertree orderings).** A pure hypertree ordering of \( H \) is a pair \( \mathcal{P} = (\prec, \lambda_\prec) \) where

- \( \prec \) is a linear ordering of \( V(H) \) and
- \( \lambda_\prec \) is a mapping, assigning to each vertex \( v \in V(H) \) an edge cover for \( \chi_\prec(v) \),

such that the following properties hold:

**P1** for all \( v \in V(H) \), \( \chi_\prec(v) \subseteq \bigcup \lambda_\prec(v) \), and

**P2** for all \( v \in V(H) \), \( B(v) \cap \bigcup \lambda_\prec(v) = \emptyset \).

The width of the pure hypertree ordering \( \mathcal{P} \) is \( \max_{v \in V(H)} |\lambda_\prec(v)| \).

Condition P1 states that \( \lambda_\prec \) assigns edge covers to the respective vertices. P2 represents the Special Condition, by defining the distinction between \( B \) and \( R \).

**Theorem 3.** The hypertree width of a connected hypergraph equals the minimum width over all its pure hypertree orderings.

We prove the theorem in Appendix A and introduce our second characterisation next.

### 3.2. Augmented hypertree orderings

This characterisation uses a different approach that allows for a more direct definition of the Special Condition in terms of linear orderings.

**Definition 4 (Augmented hypertree orderings).** An augmented hypertree ordering of \( H \) is a triple \( \mathcal{A} = (\prec, \lambda_\prec, \equiv) \) where

- \( \prec \) is a linear ordering of \( V(H) \),
- \( \lambda_\prec \) is a mapping that assigns each \( v \in V(H) \) a set \( \lambda_\prec(v) \subseteq E(H) \), and
- \( \equiv \) is an equivalence relation on \( V(H) \)

such that the following conditions hold:

**A1** For each vertex \( v \in V(H) \), \( \lambda_\prec(v) \) is an edge cover of

\[
\chi_\prec(v) := \{ v \} \cup \{ w \mid (v, w) \in A \} \cup \{ w \in V(H) \mid v \equiv w \}.
\]

**A2** For any two vertices \( u, v \in V(H) \) with \( (u, v) \in A^* \) and any \( e \in \lambda_\prec(v) \), it holds that \( (e \setminus \chi_\prec(v)) \cap \chi_\prec(u) = \emptyset \).

**A3** For all vertices \( u, v, w \in V(H) \), such that \( u \equiv w \) and \( (u, v), (v, w) \in A^* \), it holds that \( u \equiv v \).

The width of the augmented hypertree ordering \( \mathcal{A} \) is \( \max_{v \in V(H)} |\lambda_\prec(v)| \).
Table 1

<table>
<thead>
<tr>
<th>Vars</th>
<th>Range</th>
<th>Semantics</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a_{i,j})</td>
<td>(1 \leq i &lt; j \leq n)</td>
<td>(v_i &lt; v_j)</td>
<td>(C)</td>
</tr>
<tr>
<td>(a^*_{i,j})</td>
<td>(1 \leq i, j \leq n, i \neq j)</td>
<td>(v_i &lt; v_j)</td>
<td>Shorthand</td>
</tr>
<tr>
<td>(w_{i,k})</td>
<td>(1 \leq i \leq n, 1 \leq k \leq m)</td>
<td>(e_k \in \lambda_e(\tau_e(v_i)))</td>
<td>(m - n)</td>
</tr>
<tr>
<td>(a_{i,j})</td>
<td>(1 \leq i, j \leq n, i \neq j)</td>
<td>((v_i, v_j) \in \mathcal{A})</td>
<td>(n \cdot (n - 1))</td>
</tr>
<tr>
<td>(b_{i,j})</td>
<td>(1 \leq i, j \leq n, i \neq j)</td>
<td>((v_i, v_j) \in \mathcal{A}^*)</td>
<td>(n \cdot (n - 1))</td>
</tr>
<tr>
<td>(r_{i,j})</td>
<td>(1 \leq i, j \leq n, i \neq j)</td>
<td>(v_i \in \mathcal{R}(v_j))</td>
<td>(n \cdot (n - 1))</td>
</tr>
<tr>
<td>(t_{i,j})</td>
<td>(1 \leq i, j \leq n, i \neq j)</td>
<td>(T_4) permits (v_i \in \bigcup \lambda_e(\tau_e(v_j)))</td>
<td>(n \cdot (n - 1))</td>
</tr>
<tr>
<td>(e_{i,j})</td>
<td>(1 \leq i &lt; j \leq n)</td>
<td>(v_i = v_j)</td>
<td>(C)</td>
</tr>
<tr>
<td>(e^*_{i,j})</td>
<td>(1 \leq i, j \leq n, i \neq j)</td>
<td>(v_i = v_j)</td>
<td>Shorthand</td>
</tr>
</tbody>
</table>

Theorem 5. The hypertree width of a connected hypergraph equals the minimum width over all its augmented hypertree orderings.

We prove this theorem in Appendix B.

The equivalence relation directly addresses the problem of accumulating hyperedges. Condition A1 redefines bags as \(\chi_m\), based on \(\equiv\), while Condition A3 ensures that T3 still holds with the definition of \(\chi_m\). With this new definition in hand, the Special Condition can be expressed as Condition A2. Given that \(A^*\) expresses the descendancy relation in the canonical tree, Condition A2 restates T4, replacing the use of the descendancy relation with \(A^*\). This definition allows having already eliminated vertices in a bag and we can thereby deal with accumulating hyperedges. In Fig. 1, the accumulating hyperedges would be solved by setting \(a \equiv b \equiv c\).

Next, we use our characterisations to propose the corresponding encodings.

4. Encodings

In this section, we utilize the new characterisations of hypertree width for two new SAT/MaxSAT/SMT encodings. In the basic setup, we follow closely the SAT encoding for treewidth as proposed by Samer and Veith [32], variants of which served as the basis for several other encodings, for treewidth and other width measures [2,5,13,28,35]. Table 1 gives an overview of the variables used. We note in passing, that the challenge of finding a characterisation of hypertree width that can be put on top of the Samer-Veith encoding of treewidth was our original motivation to pursue this work.

Again, we assume a given connected hypergraph \(H = (V, E)\) with \(n\) vertices \(v_1, \ldots, v_n \in V(H)\), \(m\) edges \(e_1, \ldots, e_m \in E(G(H))\), and a bound \(W\) on the hypertree width. The task is to produce a propositional formula \(F(H, W)\) in Conjunctive Normal Form, which is satisfiable if and only if \(htw(H) \leq W\). The construction will allow us to efficiently transform a satisfying assignment for \(F(H, W)\) into a hypertree ordering, and in turn into a hypertree decomposition of width \(\leq W\).

4.1. Base encoding

First, we review the basic setup following Samer and Veith’s [32] treewidth encoding. The variables \(o_{i,j}\) define \(<\), where \(o_{i,j}\) is true if and only if \(v_i < v_j\). We use the shorthand \(o^*_{i,j}\):

\[
o^*_{i,j} := \begin{cases} o_{i,j} & \text{if } i < j; \\ \lnot o_{j,i} & \text{otherwise}. \end{cases}
\]

The following set of clauses establishes transitivity:

\[
\bigwedge_{1 \leq i,j,k \leq n\atop i \neq j \neq k \neq i} \lnot o^*_{i,j} \lor \lnot o^*_{j,k} \lor o^*_{i,k}.
\]

The variables \(a_{i,j}\) define \(A\), where \(a_{i,j}\) is true if and only if \((v_i, v_j) \in \mathcal{A}\). Initially, \(A\) consists of the edges from \(E(G(H))\) in forward direction, expressed by the following three sets of clauses:

\[
\bigwedge_{1 \leq j \leq n\atop i \neq j} \lnot a_{i,j} \lor a_{j,i}, \quad \bigwedge_{(v_i, v_j) \in E(G(H))} \lnot o^*_{i,j} \lor a_{i,j}, \quad \bigwedge_{(v_i, v_j) \in E(G(H))} \lnot a^*_{j,i} \lor a_{j,i}.
\] *(1)*

The following clauses add fill-in edges and complete the definition of \(A\):

\[
\bigwedge_{1 \leq i,j,k \leq n\atop i \neq j \neq k \neq i} \lnot o^*_{j,k} \lor \lnot a_{i,j} \lor \lnot a_{i,k} \lor a_{j,k}.
\]
Variables $b_{i,j}$ express $A^*$, where $b_{i,j}$ is true if and only if $(v_i, v_j) \in A^*$. The following clauses initialize $A^*$ with $A$ and enforce transitivity:

$$
\bigwedge_{1 \leq i, j \leq n, \ i \neq j} \neg a_{i,j} \lor b_{i,j},
\bigwedge_{1 \leq i, j, k \leq n, \ i \neq j \neq k} \neg a_{j,k} \lor \neg b_{i,j} \lor b_{i,k}.
$$

We conclude the base encoding, by expressing P1/A1 and thereby $\lambda_\prec /\lambda_\prec$. This follows the encoding by Berg et al. [5] for generalized hypertree decompositions. The weight variable $w_{i,k}$ is true if and only if $e_k \in \lambda_\prec (v_i)$. For the SMT encoding, instead of Boolean variables, we use integer valued variables. These integer variables can take the values 1 and 0, corresponding to true and false. The following clauses express P1/A1, or $\bigcup \lambda_\prec (v) \supseteq \chi_\prec (v)$:

$$
\bigwedge_{1 \leq i \leq n, \ v_i \in \mathcal{E}} \bigvee_{v_j \in \mathcal{E}} w_{i,k},
\bigwedge_{1 \leq i, j \leq n, \ i \neq j} \bigvee_{e_k \in \mathcal{E}} \bigwedge_{v_j \in \mathcal{E}} \neg a_{i,j} \lor \bigvee_{v_j \in \mathcal{E}} w_{i,k}.
$$

For the SMT encoding, we replace the disjunctions over $w_{i,k}$ by a constraint stating that the sum over the weight variables must be at least 1.

4.1. Cardinality constraints

It remains to add cardinality constraints that encode that each bag has an edge cover of size $\leq W$. We use our encodings with different types of solvers, each of them requiring a slightly different method of encoding cardinality constraints. We discuss our approaches for SAT, MaxSAT, and SMT solvers.

SAT solvers When using a SAT solver, we express the cardinality constraints directly in propositional logic (as by Samer and Veith [32] in the context of treewidth). In our SAT encoding, we use totalizer cardinality constraints [1]. These performed best among all cardinality constraints provided by PySAT [26] and are inherently compatible with incremental SAT solving [29].

MaxSAT solvers While the SAT solver decides whether $htw(H) \leq W$, a MaxSAT solver can find the minimum $W$ such that the encoding is satisfiable, i.e., the solver can directly compute $htw(H)$. For the MaxSAT solver, we use the totalizer cardinality constraints as in the SAT encoding. These constraints define the variables $c_{i,k}$, $1 \leq i \leq n$, $1 \leq k \leq W$, where $c_{i,k}$ is true if $|\lambda_\prec (v_i)| \geq k$. Therefore, the encoding so far constitutes the hard clauses. Additionally, we add variables $m_k$ for $1 \leq k \leq W$, where $m_k$ is true if the hypertree width is less than $k$ (as by Berg et al. [5] in the context of generalized hypertree width). We ensure the semantics of $m$ by adding for $1 \leq i \leq n$, $1 \leq k \leq W$, the clauses $\neg m_k \lor \neg c_{i,k}$. Finally, we let the solver minimize the width by adding for $1 \leq k \leq W$ the soft clauses $m_k$. The MaxSAT solver now finds a model satisfying all hard clauses and setting as many variables $m_k$ to true as possible and thereby finds the model corresponding to a hypertree decomposition of minimum width.

SMT solvers An SMT solver allows for more abstract handling of cardinalities by the algebraic theory solver within the SMT approach (as used by Fichte et al. [13] in the context of fractional hypertree width). In our SMT encoding, we can encode the necessary constraints for $i \leq n$ directly as $\sum_{e_k \in \mathcal{E}} w_{i,k} \leq W$. Additionally, SMT solvers can, similar to MaxSAT solvers, automatically search for the minimum $W$ such that the encoding is satisfiable.

This concludes the base encoding. We now discuss the encodings of the pure and augmented characterisations of hypertree orderings, respectively.

4.2. Pure encoding

This encoding expresses property P2 in two parts: (i) define $R$, and (ii) use this definition to express the Special Condition. We discuss this encoding and a possible improvement in this section.

The repairable T4-violations $R$ are represented by the variables $r_{i,j}$, where $r_{i,j}$ is true if and only if for every vertex $v_k$ on the arc-path between $v_i$ and $v_j$ it holds that $\lambda_\prec (v_i) \subseteq \lambda_\prec (v_k)$. We first encode that every hyperedge in the edge cover of $v_i$ must also occur in the edge cover of $v_j$, thereby ensuring $\lambda_\prec (v_i) \subseteq \lambda_\prec (v_j)$, expressed by the clauses

$$
\bigwedge_{1 \leq i, j \leq n, \ 1 \leq k \leq m, \ i \neq j} \neg r_{i,j} \lor \neg w_{j,k} \lor w_{i,k}.
$$

This property must hold along the entire arc-path. We express this by stating that the property only holds for $v_k$ if it also holds for all predecessors $v_j \in P(v_i, v_k)$ along the path:

$$
\bigwedge_{1 \leq i, j, k \leq n, \ i \neq j \neq k} \neg b_{i,j} \lor \neg b_{j,k} \lor \neg r_{i,k} \lor r_{i,j}.
$$
The Special Condition can be concisely stated with these two sets of variables. Whenever a vertex \( v_j \) is arc-reachable from vertex \( v_i \), the edge cover of \( v_j \) must not use any hyperedge containing \( v_i \). The only exception is the case, when the edge cover of every vertex on the arc-path is a superset of \( v_i \)'s edge cover. The following clauses express the Special Condition:

\[
\bigwedge_{1 \leq i, j \leq n, e_k \in E(H)} \neg b_{i, j} \lor r_{j, i} \lor \neg w_{j, k}.
\]

4.2.1. Improving the encoding of the special condition

The number of variables can be further reduced by encoding P2 in a less direct way. This improvement uses two ideas. The first idea is to not encode the forbidden, but the permitted vertices, i.e., \( V(H) \setminus B(v) \). This makes it possible to use only one set of variables that combines \( s_{i, j} \) and \( r_{i, j} \). The other idea is to restrict the vertices we consider for \( R \).

We use variables \( \ell_{i, j} \), where \( \ell_{i, j} \) is true if and only if \( v_i \) is permitted in the edge cover of \( v_j \). The first set of clauses restricts the use of hyperedges in the edge covers as before:

\[
\bigwedge_{1 \leq i, j \leq n, e_k \in E(H)} \ell_{i, j} \lor \neg w_{j, k},
\]

and the second set of clauses ensures that the allowed property holds along arc-paths:

\[
\bigwedge_{1 \leq i, j, k \leq n, i \neq j \neq k \neq i} \neg a_{i, k} \lor \ell_{i, j} \lor \neg \ell_{i, k}.
\]

It remains to encode the conditions that define when a vertex is forbidden. Due to the following result, we can restrict the scope of these arc-paths.

**Proposition 6.** Let \( u \in V(H) \) and \( v \) be the \( \prec \)-largest vertex, such that \((u, v) \in A\). For every vertex \( w \) such that \( v \prec w \), we can remove any hyperedge containing \( u \) from \( \lambda_{\prec}(w) \).

**Proof.** Consider vertices \( u, v, \) and \( w \) as in the proposition. If P2 holds before the removal of the hyperedge, it does so afterwards, as the intersection can only become smaller. It remains to show that P1 holds.

From the definition of \( A \), we know that for each vertex \( v' \) that is adjacent to \( u \), it holds that either \((u, v') \in A \) or \((v', u) \in A \). \( w \) is therefore not adjacent to \( u \). Furthermore, there is no vertex \( w' \) such that \( w \prec w' \) and \((u, w') \in E(G(H)) \), as \( v \) is the \( \prec \)-largest vertex with an arc from \( u \). Therefore, for all \( w' \) such that \( v \prec w' \), it holds that \( w' \) is not adjacent to \( u \).

This implies that no vertex in \( \chi_{\prec}(w) \) is adjacent to \( u \), and we can safely remove any hyperedge containing \( v \) from \( \lambda_{\prec}(w) \), as \( \chi_{\prec}(w) \) will still be covered. \( \square \)

Using this result, we can discard all vertices that are not direct successors of \( v_j \). We add the clauses

\[
\bigwedge_{1 \leq i, j \leq n, e_k \in E(H)} \neg a_{i, j} \lor \neg a_{j, k} \lor \neg a^{e}_{i, k} \lor a_{i, k} \lor \neg \ell_{i, k}.
\]

It remains to check the subset property. As before, we add the clauses

\[
\bigwedge_{1 \leq i, j \leq n, 1 \leq k \leq m, i \neq j} \neg a_{i, j} \lor \neg \ell_{i, j} \lor \neg w_{i, k} \lor w_{j, k}.
\]

This concludes the improved pure encoding. The improvement halves the number of variables and slightly reduces the number of clauses. Next, we discuss our second encoding.

4.3. Augmented encoding

This encoding follows Definition 4 for augmented hypertree orderings. This encoding has three parts: (i) definition of \( \equiv \), (ii) extending \( A \) to \( A^\equiv \), and (iii) verifying the Special Condition.

The variables \( e_{i, j} \) encode the equivalence relation \( \equiv \), where \( e_{i, j} \) is true if and only if \( v_i \equiv v_j \). We define the shorthand \( e^{e}_{i, j} \) such that
\[ e_{i,j}^* := \begin{cases} e_{i,j} & \text{if } i < j; \\ e_{j,i} & \text{otherwise.} \end{cases} \]

The following clauses ensure the transitivity of \( \equiv \) and property A3: if two vertices are equivalent, they are equivalent to all the vertices on the arc-path between them:

\[
\bigwedge_{1 \leq i, j, k \leq n, i \neq j, k} -e_{i,j}^* \lor -e_{j,k}^* \lor e_{i,k}^*, \quad \bigwedge_{1 \leq i, j, k \leq n, i \neq j, k} -e_{i,k}^* \lor -b_{i,j} \lor -b_{j,k} \lor e_{i,j}^*.
\]

We adapt \( A \) to \( A^\infty \) next. This requires a change of the clauses in Equation (1), allowing backward-arcs within equivalence classes:

\[
\bigwedge_{1 \leq i, j \leq n, i \neq j} e_{i,j}^* \lor -a_{i,j} \lor -a_{j,i}, \quad \bigwedge_{(v_i,v_j) \in E(G(H))} e_{i,j}^* \lor -a_{i,j}^* \lor a_{i,j}, \quad \bigwedge_{(v_i,v_j) \in E(G(H))} e_{i,j}^* \lor -a_{i,j}^* \lor a_{j,i}.
\]

The backward-arcs within equivalence classes are enforced using the following clauses:

\[
\bigwedge_{1 \leq i, j \leq n, i \neq j} -e_{i,j}^* \lor a_{i,j}
\]

Verifying the Special Condition is now possible with the following set of clauses:

\[
\bigwedge_{1 \leq i, j \leq n, e \in E(H), i \neq j, v_i \neq v_j} -b_{i,j} \lor -w_{j,k} \lor e_{i,j}^*.
\]

This completes the augmented encoding.

4.4. Encoding comparison

The correctness of the encodings follows by construction and Theorems 3 and 5. Obviously, the SAT/MaxSAT/SMT formula \( F(H, W) \) can be constructed in polynomial time, given \( H \) and \( W \), hence we arrive at the following result.

**Theorem 7.** Given a hypergraph \( H \) with \( n \) vertices and \( m \) hyperedges and an integer \( W \), we can construct in time polynomial in \( n + m + W \) a SAT/MaxSAT/SMT formula \( F(H, W) \) which is satisfiable if and only if \( htw(H) \leq W \). In case of the MaxSAT encoding, this holds for the hard clauses. Additionally, the model for the optimal solution to the MaxSAT encoding corresponds to a hypertree decomposition of width \( htw(H) \) (see Section 4.1.1).

Asymptotically, the number of clauses is in \( \Theta(n^3 + mn^2) \) for the pure encoding and in \( \Theta(n^3 + mn) \) for the augmented encoding, while the number of variables is in \( \Theta(n^3) \) for both. These asymptotics are similar to other encodings for structural decompositions [13,32,33]. However, the actual constants have a significant influence on performance and scalability of the encoding.

Table 2 shows how many variables and clauses are used by the different encodings. On top of the shown numbers, variables and clauses are added to encode the cardinality constraints, which depends on the method chosen. The different hypertree ordering encodings approximately double the size of the base encoding. The augmented encoding requires the most clauses, but is not as dependent on the number of primal graph edges as the pure encoding. How many clauses the pure encoding requires, depends strongly on \( m \). Since \( m \) is bound by \( n^2 \), the encoding can become asymptotically very large for dense graphs, but is small for sparse graphs.

Introducing auxiliary variables can circumvent this problem. Instead of \( w_{i,e} \), Equations (2) and (3) use variables \( c_{i,j} \), where \( c_{i,j} \) is true if and only if \( v_j \in \lambda<(v_i) \). This requires \( \frac{m^2}{10n^3} \) many clauses instead of \( mn \cdot (n - 1) \). This method also
adds $n^2$ variables $c_{ij}$, and $2mn + n^2$ clauses for its definition. The result is an encoding that has better asymptotic bounds for dense graphs and requires slightly more variables and clauses than the augmented encoding.

Next, we discuss our empirical evaluation of these encodings.

5. Experiments

Results\(^3\) and code\(^4\) of the experiments are available online.

**Experimental setup** We implemented the encodings presented in Section 4 using PySAT 1.6.0.\(^5\) We compared the encoding-based approaches with the state-of-the-art decomposer new-det-k-decomp\(^6\) by Fischl et al. [14], which implements an XP-algorithm based on dynamic programming over separators.\(^7\)

We used the following solvers. Optimathsat 1.7.2\(^7\) [36] was used for the SMT encodings, as it performed better than z3 4.8.9. We tried the SAT solvers provided by PySAT [26], as well as KisSAT 2.0.1\(^8\) [7] for SAT instances. In our experiments, we used MapleChrono as it performed best. MaxSAT instances were solved by MaxHS 4.0.0\(^10\) [6], the winner of the MaxSAT Evaluation 2020.\(^11\) For the SAT encoding, we start at an initial heuristically computed width. This width is then decremented until the formula is unsatisfiable. We used servers with two Intel Xeon E5540 CPUs, each running at 2.53 GHz per core. The servers used Ubuntu 18.04. Each run was limited to six hours and 32 GB RAM. In the results each non-solved instance is counted as six hours towards the total runtime.

**Benchmark instances** We tested against the full set of Hyperbench\(^12\) instances. Hyperbench consists of instances gathered from various database queries and constraint satisfaction problem instances \([4,9,10,16,31]\), for which hypertree width is of practical relevance. The instances of the PACE 2019 competition form a subset of the Hyperbench instances. We removed all disconnected instances, as by Proposition 1 we could solve them component wise, hence their inclusion would skew those results that are based on the instance size. We ran each configuration against the remaining 3008 instances. 2396 of these instances were solved by any solver configuration and 1261 were solved by all solver configurations. We present the results for the 2396 solved instances.

5.1. Experimental results

The aim of the experiments was to see how encodings based on different characterisations of hypertree width perform in comparison, and not to determine the fastest exact method for computing the hypertree width.

In Table 3 we see a summary of how many instances the different configurations and solvers were able to solve. This summary, grouped by the hypertree width of the instances, is shown in Table 4. Further, Figs. 3 and 4 show how runtime and memory usage develop, as the instances become harder to solve.

**Encodings** The pure encoding performed slightly better than the augmented encoding in terms of solved instances by a single solver type. The difference is very small, also in terms of uniquely solved instances: using a MaxSAT solver, the

<table>
<thead>
<tr>
<th>Configuration/Solver</th>
<th># Solved</th>
<th>Time [s]</th>
<th>Unique</th>
</tr>
</thead>
<tbody>
<tr>
<td>Augmented (MAX)</td>
<td>2099 (88%)</td>
<td>8195.7</td>
<td>13</td>
</tr>
<tr>
<td>Augmented (SAT)</td>
<td>1886 (78%)</td>
<td>10295.7</td>
<td>4</td>
</tr>
<tr>
<td>Augmented (SMT)</td>
<td>2055 (86%)</td>
<td>8593.1</td>
<td>113</td>
</tr>
<tr>
<td>Pure Improved (MAX)</td>
<td>2103 (88%)</td>
<td>8159.5</td>
<td>12</td>
</tr>
<tr>
<td>Pure Improved (SAT)</td>
<td>1820 (76%)</td>
<td>10710.3</td>
<td>0</td>
</tr>
<tr>
<td>Pure Improved (SMT)</td>
<td>1837 (77%)</td>
<td>10557.2</td>
<td>6</td>
</tr>
<tr>
<td>New-DetK</td>
<td>1478 (62%)</td>
<td>13793.4</td>
<td>14</td>
</tr>
</tbody>
</table>

---

\(^3\) https://doi.org/10.5281/zenodo.6832106.
\(^4\) https://github.com/ASchidler/hitsmt.
\(^5\) https://github.com/ASchidler/maxsmt.
\(^6\) https://github.com/TUfischl/newdetkdecomp/ (Commit c06232d).
\(^7\) Recently the algorithm has been parallelized in the solver BalancedGo [23]. We compare against new-det-k-decomp as our solution is not parallelized.
\(^8\) https://optimathsat.disi.unitn.it.
\(^9\) https://fmw.jku.at/kissat/.
\(^10\) https://github.com/fbacchus/MaxHS.
\(^12\) https://hyperbench.dbai.tuwien.ac.at/.
pure encoding solved 55 instances that the augmented encoding could not solve. Interestingly, runtime and memory usage behaved almost identical as well.

The augmented encoding was the better encoding if we would use more than one solver type in a portfolio approach. While using an SMT solver for the pure encoding had little benefit over the MaxSAT solver, it produced interesting results for the augmented encoding. Here, the SMT solver solved 112 additional instances, thereby surpassing the pure encoding in a portfolio approach together with the MaxSAT solver.

The difference in performance might come from how well the two encodings handle high-width instances. Table 4 suggests that the augmented encoding handles higher widths better. While the difference is small for MaxSAT, the gap becomes significant for the other solver types. The reason why the pure encoding did slightly better with the MaxSAT solver might stem from the handling of larger graphs, as the asymptotics suggest. Indeed, Fig. 6 shows that the pure encoding performed slightly better on instances with a large number of vertices.

In general, both encoding were able to solve large instances. As Fig. 6 shows, there is no order of magnitude, where all instances remained unsolved.

**Solver types** Overall, the MaxSAT solver performed best for both encodings and almost strictly better than the SAT solver. The memory consumption after encoding the instance is slightly higher when using a MaxSAT solver compared to the SAT solver. This is expected as the solver runs a SAT solver and a hitting set solver. However, the MaxSAT solver’s peak memory consumption on one third of the instances was lower than that of the SAT solver, as it solved the instance more efficiently.
Fig. 5. Scatter plots comparing the resource use of the single best encoding and new-det-k-decomp.

Fig. 6. Instance sizes and whether the instance could be solved by the Pure or Augmented MaxSAT encoding. The number of edges refers to the primal graph. The bar charts represent the shaded area around them. The degree of shading indicates how many instances of that size were in the instance set, with black being the most.

Fig. 7. Instance sizes and whether the instance could be solved by the Augmented MaxSAT or SMT encoding. The number of edges refers to the primal graph. The bar charts represent the shaded area around them. The degree of shading indicates how many instances of that size were in the instance set, with black being the most.

As is common with SAT encodings, the last satisfiable step was usually faster than the first unsatisfiable step. Hence, finding the optimal solution was faster than proving its optimality. The SAT solver can, therefore, provide good decompositions even faster, if optimality is not required. The SMT solver’s performance strongly depended on the encoding. For the pure encoding, it performed almost strictly worse than the MaxSAT solver. For the augmented encoding, it performed slightly worse than the MaxSAT solver. Interestingly, the number of uniquely solved instances is very high. Table 4 shows that the SMT solver excelled at instances with high hypertree width. This may be due to a different handling of cardinality constraints in the MaxSAT and SMT solver. Furthermore, Fig. 7 suggests that the SMT solver performs significantly better on large hypergraphs.

New-det-k-decomp New-det-k-decomp solved considerably fewer instances than any of our encodings with any of the different solver types. Table 4 suggests that the encodings are better at handling instances with higher width: new-det-k-decomp solves very few instances with a hypertree width of 5 or higher. This behavior is not surprising, as the nature of

---

13 Since a bug in new-det-k-decomp caused errors and wrong results for some instances, we counted each run that finished within the time and memory limit as successful. In total, for 21 instances and 3 uniquely solved instances it is not clear, whether the instances have indeed been solved by new-det-k-decomp.
the algorithm requires $\Omega(|V(H)|^{\text{htw}})$ time and space. Figs. 3 and 4 show that runtime and memory consumption behaved very similarly to the SAT solver: both requirements are very low for instances of small width, until they almost instantly exceed the limits. New-det-k-decomp exhibited this behavior approximately 400 instances before the SAT solvers do.

Fig. 5 compares the resource usage of the augmented encoding and new-det-k-decomp for the same instances. Here, most marks are far away from the diagonal. This indicates that the two methods perform very differently when given the same instance. Hence, although new-det-k-decomp performs worse overall, there are several instances where it requires fewer resources than the augmented encoding.

6. Concluding remarks

Summary We presented the first ordering-based characterisations of hypertree width that are purely based on linear elimination orderings. This characterisation provides new combinatorial insights into hypertree decompositions. We utilized the characterisations for new SAT/MaxSAT/SMT encodings and tested them on an extensive set of benchmark instances. Indeed, the new encoding clearly outperforms the state-of-the-art combinatorial algorithm for hypertree decompositions. We expect that the new ordering-based characterisations of hypertree width will also be of interest outside SAT encodings, for instance, for Branch & Bound algorithms.

Our experimental results also show interesting differences in not only how many, but which instances can be solved by our encodings and different solver types, particularly SMT and MaxSAT solvers.

Future work We hope that in the future one can build upon our ordering-based characterisations and encoding, and develop further improvements, including preprocessing and inprocessing techniques, so that optimal hypertree decompositions can be efficiently found for even larger instances.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

The data used is openly available at the sources indicated in the paper.

Acknowledgements

The authors acknowledge the support from the FWF (P32441, P36420, W1255) and the WWTF (ICT19-065).

Appendix A. Proof of Theorem 3

In this section, we establish Theorem 3 in an algorithmic way, by providing polynomial-time algorithms that transform hypertree decompositions into pure/augmented hypertree orderings of the same width, and conversely, polynomial-time algorithms that transform pure/augmented hypertree orderings into hypertree decompositions of the same width. Also for this section, let $H$ be an arbitrary connected hypergraph.

A.1. From hypertree decompositions to pure hypertree orderings

Let $\mathcal{D} = (T_D, \chi_D, \lambda_D)$ be a hypertree decomposition of $H$ of width $W$. We will translate $\mathcal{D}$ into a pure hypertree ordering $\mathcal{P}$ of width $W$.

We assume, w.l.o.g., that the following properties hold for $\mathcal{D}$:

**N1** For all $t \in V(T_D)$, $e \in \lambda_D(t)$: $e \cap \chi_D(t) \neq \emptyset$.

**N2** For all $(t, t') \in E(T_D)$, if $t'$ is the parent of $t$, then $\chi_D(t) \not\subseteq \chi_D(t')$.

We observe that N1 can be established by removing all violating hyperedges from the respective covers. N2 can be established by contracting edges between violating nodes and retaining the parent’s cover and bag. Due to this property, at least one vertex is forgotten at every node (recall the definition of “forgetting a vertex” from Section 2).

Next, we define a preorder $\preceq$ on $V(H)$ by setting $v \preceq w$ if and only if $f(v)$ is in the subtree rooted at $f(w)$ (this includes the case $f(v) = f(w)$). By letting $\prec$ be any total ordering that refines $\preceq$ and setting $\lambda_p(v) := \lambda_D(f(v))$, we define $\mathcal{P} = (\prec, \lambda_p)$. Further, let $A$ be the set of active arcs according to $\prec$ and $A^*$ its transitive closure (see Section 3).

We first show some properties of the transformation and then establish its correctness. The following two observation state basic properties of pure hypertree orderings and will be used in the following proofs:
Observation 8. If \((v, w) \in A^*\), then there is some \(u \in V(H)\) such that \((u, w) \in A\), where possibly \(u = v\).

Lemma 9. If \((u, w) \in A\) and \((u, v) \in A^*\), such that \(v < w\) then \((v, w) \in A\).

Proof. Let \(v'\) be the \(<\)-smallest vertex such that \((u, v'), (v', v) \in A^*\) holds. By the definition of \(A\), \((u, v') \in A\) and therefore \((v', w) \in A^*\). The hypothesis follows inductively by using \(v'\) instead of \(u\). \(\square\)

Lemma 10. If \((v, w) \in A\) and \([v, w] \notin E(G(H))\), then there is some \(u \in V(H)\) such that \([u, w] \in E(G(H))\) and \(v\) is arc-reachable from \(u\).

Proof. Given the premise, there must be a \(u' \in V(H)\) such that \((u', w) \in A\) and \((u', v) \in A\). Either \([u', w] \in E(G(H))\) or there must be a vertex \(u'' \in V(H)\) such that \((u'', w) \in A\) and \((u'', u') \in A\). Since \(u'' < u' < v\), this process moves towards the beginning of \(<\) in a strictly monotonous fashion. We therefore arrive at a vertex \(u\) such that \([u, w] \in E(G(H))\) (Observation 8). Since we arrived at \(u\) following an arc-path, \(v\) is arc-reachable from \(u\). \(\square\)

Lemma 11. If \((u, w) \in A^*\), then \(f(u)\) is in the subtree rooted at \(f(w)\).

Proof. From the premise, we know that \(w\) is arc-reachable from \(u\). We show the desired property by induction on \(|\{v \mid (u, v) \in A^*, (v, w) \in A^*\}|\), the length of the arc-path.

Base Case: The path has length 0. In this case the premise implies that \([u, w] \in E(G(H))\). Therefore, \(T_2\) implies that \(u\) and \(w\) must occur in a bag together and due to \(T_3\) it holds that \(u \in \chi_D(f(w))\) or \(w \in \chi_D(f(u))\). Therefore, by the definition of \(<\), it holds that \(w \in \chi_D(f(u))\), as otherwise \(w < u\) which would contradict the premise that \((u, w) \in A^*\). The hypothesis is implied by \(w \in \chi_D(f(u))\) and \(T_3\).

Induction Step: The path has length \(i + 1\). By the definition of \(A^*\) there exists a \(v\) such that \((u, v) \in A^*\) and \((v, w) \in A^*\). By the induction hypothesis, \(f(v)\) is in the subtree rooted at \(f(w)\) and \(f(u)\) in the subtree rooted at \(f(v)\), consequently the induction step holds. \(\square\)

Lemma 12. If \((v, w) \in A\), then \(w \in \chi_D(f(v))\).

Proof. Case 1: \([v, w] \in E(G(H))\). By \(T_2\), there has to be a \(t \in V(T_D)\) such that \([v, w] \subseteq \chi_D(t)\). Due to \(T_3\), \(w\) has to occur in the bag of every node on the path between \(t\) and \(f(w)\). Since \(v \in \chi_D(t)\) and by Lemma 11, \(f(v)\) is in the subtree of \(T_D\) rooted at \(f(w)\), thus \(w \in \chi_D(f(v))\) and the lemma holds.

Case 2: \([v, w] \notin E(G(H))\). By Lemma 10, there exists some \(u \in V(H)\) such that \([u, w] \in E(G(H))\) and \(v\) is arc-reachable from \(u\). By \(T_2\), there is a node \(t \in V(T_D)\) such that \([u, w] \subseteq \chi_D(t)\). Due to Lemma 11, \(f(u)\) is in the subtree of \(T_D\) rooted at \(f(v)\) and \(f(u)\) is in the subtree rooted at \(f(w)\). Since \(v < w\), it holds that \(f(v)\) is in the subtree rooted at \(f(w)\). Since \(w \in \chi_D(t)\), by \(T_3\), \(w\) must occur in all bags of the nodes on the path between \(t\) and \(f(w)\), including \(f(v)\). Therefore, the lemma follows. \(\square\)

Proposition 13. \(P\) is a pure hypertree ordering of \(H\) of width \(W\).

Proof. We first observe that we do not change any edge cover and therefore, the width does not increase.

Next we observe that \(P_1\) holds. For every vertex \(v \in V(H)\) it holds that \(v \in \bigcup \lambda_\prec(v)\) by the definition of \(\lambda_\prec\). Further, for every vertex \(w \in V(H)\) such that \((v, w) \in A\) it holds that \(w \in \chi_D(f(v))\) by Lemma 12. This implies that \(w \in \bigcup \lambda_D(f(v)) = \bigcup \lambda_\prec(v)\) and therefore, \(P_1\) holds.

Next, we show that \(P_2\) holds by contradiction. We assume for vertices \(v, w \in V(H)\) that \(v \in B(w)\) and \(v \in \bigcup \lambda_\prec(w)\).

Given the definition of \(B\) and Lemma 11, we know that \(v\) occurs in at least one of the bags in the subtree rooted at \(f(w)\). Since \(\lambda_\prec(w) = \lambda_D(f(w))\) and \(v \in \bigcup \lambda_\prec(w)\) it must hold that \(v \in \chi_D(f(w))\) due to \(T_4\). Since \(v < w\), this implies \(f(v) = f(w)\), therefore \(\lambda_\prec(v) = \lambda_\prec(w)\). Consequently, every \(u \in P(v, w)\) has the same edge cover and thereby \(v \in R(w)\), implying \(v \notin B(w)\) and contradicting the assumption. \(\square\)

A.2. From pure hypertree orderings to hypertree decompositions

We now proceed and show the other direction. The following lemma formally justifies the definition of \(R\).

Lemma 14. Let \(D = (T_D, \chi_D, \lambda_D)\) be a hypertree decomposition and let \(t, t' \in V(T_D)\), such that \([t, t'] \in E(T_D)\) and \(t'\) is the parent of \(t\). If \(\lambda_D(t) \subseteq \lambda_D(t')\), then we can add all vertices from \(\chi_D(t)\) to \(\chi_D(t')\) and \(D\) remains a valid hypertree decomposition.
Proof. Since T1–T4 hold before the transformation, we show that they also hold after the transformation. Since we did not remove any vertices from the bags, T1 and T2 still hold. Since t and t’ are adjacent in the tree, T3 also holds. Since by adding vertices to the bag, we can only reduce the number of omitted vertices, T4 also holds. Finally, since \( \lambda_D(t) \) contains at least the hyperedges in \( \lambda_D(t) \), it remains a valid edge cover. □

Corollary 15. Let \( D \) be a hypertree decomposition \( (T_D, \chi_D, \lambda_D) \) and let \( t, t’ \in V(T_D) \), such that \( t \) is in the subtree rooted at \( t’ \). If for every node \( t’ \) on the path between \( t \) and \( t’ \) (including \( t’ \)) it holds that \( \lambda_D(t) \subseteq \lambda_D(t’) \), then we can add all vertices in \( \chi_D(t) \) to \( \chi_D(t’) \) and \( D \) remains a valid hypertree decomposition.

Given a pure hypertree ordering \( P = (\prec, \lambda_\prec) \) of \( H \), we define \( D = (T_\prec, \chi_D, \lambda_D) \), where

- \( T_\prec \) is the canonical tree,
- \( \lambda_D(T_\prec(v)) = \lambda_\prec(v) \) for all \( v \in V(H) \), and
- \( \chi_D(T_\prec(v)) := \chi_\prec(v) \cup \bigcup_{u \in R(v)} \chi_\prec(u) \) for all \( v \in V(H) \).

This definition implies that for every \( v \in V(H) \), \( f(v) = \tau_\prec(v) \).

Lemma 16. If \( v \in \chi_D(T_\prec(u)) \) and \( u < v \), then \( v \in \chi_\prec(u) \) and \( (u, v) \in A \).

Proof. Since \( v \in \chi_D(T_\prec(u)) \) either (i) \( (u, v) \in A \) or (ii) \( (u’, v) \in A \) with \( u’ \in R(u) \). Case (i) implies that \( v \in \chi_\prec(u) \) by definition of \( \chi_\prec \). Case (ii) implies that \( u \) is arc-reachable from \( u’ \). Let \( u’’ \) be the successor of \( u’ \) in the arc-path from \( u’ \) to \( u \). Since \( (u’, v) \in A \) and \( (u’, u’’) \in A \) it follows that \( (u’’, v) \in A \). The same argument holds for \( u’’ \) and repeating the process eventually shows the desired property for \( u \). □

Corollary 17. If \((u, w) \in A, (u, v) \in A^* \), and \( v < w \), then \( (v, w) \in A \).

Proposition 18. \( D = (T_\prec, \chi_D, \lambda_D) \) is a hypertree decomposition of \( H \) of width \( W \).

Proof. By definition of the canonical tree, \( T_\prec \) is rooted at \( \tau_\prec(r) \in V(T_\prec) \), where \( r \in V(H) \) is the \( \prec \)-largest vertex. We argue that every node, except \( \tau_\prec(r) \), has exactly one parent, i.e., that \( T_\prec \) is indeed a tree. Since \( H \) is connected, for each vertex \( u \in V(H) \), except \( r \), there exists a vertex \( v \in V(H) \) such that \( (u, v) \in A \). Due to the definition of \( E(\tau_\prec) \) this implies that \( \tau_\prec(u) \) is connected to exactly one \( \tau_\prec(v) \) such that \( u < v \). Therefore, \( \tau_\prec(v) \) is the parent of \( \tau_\prec(u) \) and \( T_\prec \) is connected. Since Property P1 holds, it follows that for every \( \tau_\prec(v) \in V(T_\prec) \), \( R(v) \) only contains vertices covered by \( \lambda_\prec(v) \). Therefore, the preconditions of Corollary 15 are met and \( \lambda_D(v) \) is a valid edge cover for \( \chi_D(v) \).

Conditions T1 and T2 hold as well, as we have the following for each \( e \in E \). Let \( u \) be the \( \prec \)-smallest vertex in \( e \), then for each \( v \in e \setminus \{u\} \) it holds that \( (u, v) \in A \). Therefore, by the construction of \( \chi_D, e \subseteq \chi_D(T_\prec(u)) \).

Next we show that T3 holds. Let \( u, w \in V(H) \) be arbitrary vertices, such that \( w \in \chi_D(T_\prec(u)) \). Whenever \( w < u \) and therefore \( w \notin \chi_\prec(u) \), T3 holds on the path from \( \tau_\prec(w) \) to \( \tau_\prec(u) \) by construction of \( R \) and \( \chi_D \). We therefore assume that \( u < w \). It remains to show that for each node \( \tau_\prec(v) \in V(T_\prec) \) on the path between \( \tau_\prec(u) \) and \( \tau_\prec(w) \) it holds that \( w \in \chi_D(T_\prec(v)) \). We proceed by induction on the position of \( \tau_\prec(v) \) on the path from \( \tau_\prec(u) \) to \( \tau_\prec(w) \).

Base Case: The first edge on the path is \([\tau_\prec(u), \tau_\prec(v)] \in E(T_\prec) \). By Lemma 16 it holds that \( (u, w) \in A \). It then follows from Corollary 17 that \( (v, w) \in A \), thereby showing the base case.

Induction Step: Consider an arbitrary edge \([\tau_\prec(u), \tau_\prec(v)] \in E(T_\prec) \) on the path between \( \tau_\prec(u) \) and \( \tau_\prec(v) \). Since we know that \( w \in \chi_D(T_\prec(v')) \), we can use the same argument as in the base case to show that \( w \in \chi_D(T_\prec(v'')) \). Therefore, the hypothesis and T3 hold.

Next we prove that T4 holds by contradiction. Assume vertices \( u, v, w \in V(H) \), such that \( v \) is omitted at \( \tau_\prec(w) \), \( v \in \chi_D(T_\prec(u)) \) and \( \tau_\prec(u) \) is a descendant of \( \tau_\prec(w) \) in \( T \). Since \( v \notin \chi_D(T_\prec(u)) \), either \( (u, v) \in A \) or \( (u', v) \in A \), where \( u' \in R(u) \). Both cases imply that either (i) \( (v, w) \in A^* \) or (ii) \( (w, v) \in A^* \). Case (i) implies that \( v \in R \) as otherwise P2 would be violated. This contradicts the assumption that \( v \notin \chi_D(T_\prec(v)) \). Case (ii) implies by Lemma 16 that \( (u, v) \in A \). Therefore, by Corollary 17, \( (w, v) \in A \), contradicting the initial assumption that \( v \) is omitted at \( \tau_\prec(w) \). Therefore, T4 holds.

Since T is a tree, \( \lambda_D \) assigns edge covers, and T1–T4 hold, \( D \) is indeed a hypertree decomposition for \( H \) of width \( W \). □

Appendix B. Proof of Theorem 5

Similar to the previous section, we prove this theorem in an algorithmic way. As before, let \( H \) be an arbitrary connected hypergraph.
For convenience we write
\[ A^\equiv := A \cup \{ (u, v) \mid u \equiv v \}, \]
which gives
\[ \chi^\equiv(v) = \{ v \} \cup \{ w \mid (v, w) \in A^\equiv \}. \]

### B.1. From hypertree decompositions to augmented hypertree orderings

Let \( D = (T_D, \chi_D, \lambda_D) \) be a hypertree decomposition of \( H \) of width \( W \). We define \( A = (\prec, \lambda_D^\equiv, \equiv) \) as follows:

- We let \( \equiv \) be the equivalence relation where any two vertices \( u, v \in V(H) \) are equivalent (in symbols \( u \equiv v \)), if \( f(u) = f(v) \).
- Let \( \prec^e \) be the partial order where \( u \prec^e v \) if \( f(u) \) is a descendant of \( f(v) \) in \( T_D \), for any \( u, v \in V(H) \). We let \( \prec \) be an arbitrary, but fixed, total order that refines \( \prec^e \).
- We let \( \lambda_D^\equiv(u) = \lambda_D(f(u)) \) for all \( u \in V(H) \).

**Proposition 19.** \( A \) is an augmented hypertree ordering of \( H \) of width \( W \).

**Proof.** We show that \( A1 \) holds, i.e., that \( \lambda_D^\equiv(u) \) contains edge covers for all bags in \( \chi^\equiv \). For this purpose, we assume \( w.l.o.g. \), that the following holds for \( T_D \): for any two nodes \( t, t' \in V(T_D) \) such that \( [t, t'] \in E(T_D) \), it holds that \( \chi_D(t) \setminus \chi_D(t') \neq \emptyset \) and \( \chi_D(t') \setminus \chi_D(t) \neq \emptyset \), i.e., neither of the two sets \( \chi_D(t) \) and \( \chi_D(t') \) is a subset of the other. This can be achieved by contracting violating edges \([t, t']\) and retaining the larger bag. This does not affect the validity of the decomposition, as \( T1, T2, T3 \) and \( T4 \) still hold. This assumption implies that at each node in \( T \) at least one vertex is forgotten (recall, again, the definition of “forgetting a vertex” from Section 2).

We will argue by case distinction that, given an arbitrary vertex \( u \), it holds that \( \chi^\equiv(u) \subseteq \chi_D(f(u)) \). A vertex \( v \) is contained in \( \chi^\equiv(u) \) for at least one of the four reasons:

(i) \( u = v \),
(ii) \( u \equiv v \),
(iii) \( [u, v] \in E(G(H)) \), or
(iv) \( (u, v) \) is a fill-in edge in \( A^\equiv \).

**Case (i):** the hypothesis \( \chi^\equiv(u) \subseteq \chi_D(f(u)) \) holds, as \( u \in \chi_D(f(u)) \).

**Case (ii):** similarly to Case (i), the hypothesis holds, as by definition \( f(u) = f(w) \).

For the remaining cases, we assume that the first two cases do not prevail.

**Case (iii):** by \( T2 \), there exists a bag \( \chi_D(t) \) such that \( u, v \in \chi_D(t) \). By \( T3 \) either \( f(u) \) is a descendant of \( f(v) \) or \( f(v) \) a descendant of \( f(u) \). Since \( v \in \chi^\equiv(u) \) and we eliminated Cases (i) and (ii) we know that \( u \prec v \) and therefore by definition \( f(u) \) is a descendant of \( f(v) \). By \( T3 \) this implies that \( v \in \chi_D(f(u)) \).

**Case (iv):** this case follows similarly: from the definition of \( A^\equiv \) we know that \( u \prec v \). Following the definition of fill-in edges, any fill-in edge \( (u, v) \) requires (possible fill-in) edges \((w, u)\) and \((w, v)\). From this we also know that \( w \prec u \prec v \) and by definition \( f(w) = f(u) \), or \( f(w) \) is a descendant of \( f(u) \). Since fill-in edges require other edges to the same vertex, there has to be at least one \((w', v)\) such that \((w', v) \in E(G(H)) \) and by definition \( f(w') = f(u) \) or \( f(w') \) is a descendant of \( f(u) \), where \( w' = w \) if \((w, v) \in E(G(H)) \). Now the desired property \( v \in \chi_D(f(u)) \) holds, either because \( w' \equiv u \), or because \( f(w') \) is a descendant of \( f(u) \). In the latter case, \( T2 \) implies that \( v \) occurs in a bag below \( f(u) \) and, finally, together with \( T3 \) we know that \( v \) is in the bag of \( f(u) \).

In order to show that \( A2 \) (the Special Condition) holds, we assume the contrary, i.e., there are two vertices \( u, v \in V(H) \) such that \((u, v) \in A^\equiv \) and there is a vertex \( w \in \bigcup \lambda_D^\equiv(v) \), such that \( w \not\in \chi_D^\equiv(v) \) and \( w \in \chi_D^\equiv(u) \). It cannot hold that \( u \equiv v \) as then, by definition, \( \chi_D^\equiv(u) = \chi_D^\equiv(v) \). Further, \( f(w) \) is a descendant of \( f(v) \), as otherwise \( v \prec w \) and \((v, w) \in A^\equiv \), due to fill-in edges, contradicting that \( w \not\in \chi_D^\equiv(v) \). This would imply that \( w \not\in \chi_D(f(v)) \), which would violate \( T4 \). Therefore \( A2 \) holds.

Since all vertices that are forgotten at the same node are equivalent and are next to each other in \( \prec \), \( A3 \) holds as well.

It remains to observe that, since we do not change any of the edge covers in \( \lambda_D \), the width remains the same. This concludes the proof of the proposition. □
B.2. From augmented hypertext orderings to hypertree decompositions

Let \( A = (\prec, \lambda^w, m) \) be an augmented hypertext ordering of \( H \) of width \( W \). We define \( D = (T_\prec, \chi_D, \lambda_D) \) where

- \( T_\prec \) is the canonical tree,
- \( \chi_D(T_\prec(v)) = \chi^w(v) \), and
- \( \lambda_D(T_\prec(v)) = \lambda^w(v) \).

**Proposition 20.** \( D \) is a hypertree decomposition of \( H \) of width \( W \).

**Proof.** The width of \( D \) is the same as the width of \( A \), as we do not change \( \lambda^w \), and \( A \) guarantees that for all \( v \in V(H) \), \( \lambda_D(T_\prec(v)) \) is an edge cover of \( \chi_D(T_\prec(v)) \).

\( T_1 \) holds as each \( v \in V(H) \) has a corresponding node \( T_\prec(v) \in V(T_\prec) \) and \( T_\prec(v)'s \) bag contains \( v \).

For \( T_2 \), assume an arbitrary \( v \in E(H) \) and the \( \prec \)-smallest vertex \( v \) in \( e \); now \( e \subseteq \chi_D(v) \) by the definitions of \( A^w \) and \( \chi^w \).

To verify Condition \( T_3 \), assume to the contrary that \( T_3 \) is violated; i.e., there exist nodes \( T_\prec(u), T_\prec(u'), T_\prec(u'') \in V(T_\prec) \), such that \( T_\prec(u) \) is on the path between \( T_\prec(u) \) and \( T_\prec(u'') \). Further, for some vertex \( v \in V(H) \) it holds that \( v \in \chi_D(u) \), \( v \in \chi_D(u') \), and \( v \notin \chi_D(u) \).

Therefore, the set of nodes \( \{ t \in E(T_\prec) \} \) \( v \in \chi_D(t) \} \) induces in \( T_\prec \) the (connected) subtrees \( T_1, \ldots, T_k \), where \( k > 1 \), since \( T_3 \) is violated. Let \( T_i \) be the subtree, such that \( T_\prec(v) \in V(T_i) \); due to \( A^w \{ T_\prec(w) \} \) \( v \in \chi_v \} \subseteq V(T_i) \). Further, let \( j \in \{ 1, \ldots, k \} \} \) \( i \) \( \tau_j \) \( \tau_j \) be the root of \( T_j \). By assumption, \( v \in \chi_D(T_\prec(\tau_j)) \), which implies by the definition of \( A^w \) that \( \tau_j, v \in A^w \) and that \( \tau_j \) is not the root of \( T_\prec \). Hence, there exists a parent \( \tau_j(p_j) \) of \( \tau_j \). By construction, \( v \notin \chi^w(p_j) \), and \( p_j \) is the \( \prec \)-smallest vertex with an incoming arc from \( r_j \), hence \( p_j \prec v \). Since \( r_j, v \in A^w \) and \( r_j, p_j \in A^w \), by the definition of \( A^w \) also \( p_j, v \in A^w \), contradicting \( v \notin \chi^w(p_j) \). Therefore \( T_4 \) holds.

This concludes the proof of the proposition. \( \square \)

The conjuctions of Propositions 19 and 20 establishes Theorem 5.

References


