Fast Parallel Hypertree Decompositions in Logarithmic Recursion Depth

GEORG GOTTLOB, University of Calabria, Italy and University of Oxford, UK
MATTHIAS LANZINGER, TU Wien, Austria and University of Oxford, UK
CEM OKULMUS, Umeå University, Sweden
REINHARD PICHLER, TU Wien, Austria

Various classic reasoning problems with natural hypergraph representations are known to be tractable if a hypertree decomposition (HD) of low width exists. The resulting algorithms are attractive for practical use in fields like databases and constraint satisfaction. However, algorithmic use of HDs relies on the difficult task of first computing a decomposition of the hypergraph underlying a given problem instance, which is then used to guide the algorithm for this particular instance. The performance of purely sequential methods for computing HDs is inherently limited, yet the problem is, theoretically, amenable to parallelisation. In this article, we propose the first algorithm for computing hypertree decompositions that is well suited for parallelisation. The newly proposed algorithm `log-k-decomp` requires only a logarithmic number of recursion levels and additionally allows for highly parallelised pruning of the search space by restriction to so-called balanced separators. We provide a detailed experimental evaluation over the HyperBench benchmark and demonstrate that `log-k-decomp` outperforms the current state of the art significantly.

CCS Concepts: • **Information systems** → *Relational database query languages*; • **Mathematics of computing** → *Hypergraphs*; • **Computing methodologies** → **Parallel algorithms;**

Additional Key Words and Phrases: Hypergraph decomposition, hypertree width, parallel algorithms

ACM Reference Format:
Georg Gottlob, Matthias Lanzinger, Cem Okulmus, and Reinhard Pichler. 2024. Fast Parallel Hypertree Decompositions in Logarithmic Recursion Depth. *ACM Trans. Datab. Syst.* 49, 1, Article 1 (February 2024), 43 pages. https://doi.org/10.1145/3638758

This work was supported by the Austrian Science Fund (FWF) project P30930-N35 and by the Vienna Science and Technology Fund (WWTF) [10.47379/VRG18013, 10.47379/NXT22018, 10.47379/ICT2201] and by the Christian Doppler Research Association (CDG) JRC LIVE. Georg Gottlob is a Royal Society Research Professor and acknowledges support by the Royal Society for the present work in the context of the project "RAISON DATA" (Project reference: RP\R1\201074). Matthias Lanzing acknowledges support by the Royal Society project "RAISON DATA" (Project reference: RP\R1\201074). The work of Cem Okulmus is supported by the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation.
Authors' addresses: G. Gottlob, Dipartimento di Matematica e Informatica, University of Calabria Via P. Bucci - Edificio 30B, 87036 Arcavacata di Rende, Italy and Department of Computer Science, University of Oxford, 7 Parks Road, Oxford OX1 3QG, UK; e-mail: georg.gottlob@sjc.ox.ac.uk; M. Lanzinger, Institute of Logic and Computation, TU Wien Favoritenstraße 9-11, 1040 Wien, Austria and Department of Computer Science, University of Oxford, 7 Parks Road, Oxford OX1 3QG, UK; e-mail: matthias.lanzinger@tuwien.ac.at; C. Okulmus, Department of Computing Science, Umeå University Universitetstorget 4, 90 187 Umeå, Sweden; e-mail: okulmus@cs.umu.se; R. Pichler, Institute of Logic and Computation, TU Wien Favoritenstraße 9-11, 1040 Wien, Austria; e-mail: pichler@dbai.tuwien.ac.at.

ACM 0362-5915/2024/02-ART1
https://doi.org/10.1145/3638758

1 INTRODUCTION

Hypertree decompositions (HDs) [24] have been demonstrated to be a valuable tool in a wide field of algorithmic applications. By way of structural decomposition of the hypergraph representation of problem instances, they induce tractable fragments for fundamental reasoning problems such as conjunctive query evaluation [24], constraint satisfaction problems [21], and related counting problems [29]. Other applications can be found in game theory, where problems such as determining Nash Equilibria [16] and combinatorial auctions [15] also become tractable in cases where HDs of bounded width exist.

In many of the listed cases, we do not only have theoretical tractability results but in fact know of algorithms that are suitable for practical applications. For example, in conjunctive query evaluation, HDs can be used for efficient reduction to an acyclic instance, which allows for linear-time solving using Yannakakis' algorithm [33]. Beyond practical algorithms, many of the listed problems are in fact known to be contained in the complexity class NC^2 [8] if a bounded width HD exists [3, 22, 23]. Importantly, problems in NC^2 are considered to be highly parallelisable [8] and thus the use of HDs in these areas can be even more attractive in parallelised and distributed scenarios. The promising theoretical properties of hypertree decompositions have also been experimentally verified. Implementations in specialised database systems have demonstrated the applicability of HDs in query evaluation by using them (and closely related variants), especially on difficult instances where current heuristic-based systems struggle [1, 13, 14].

Despite these desirable properties and a demand for worst-case guarantees in various potential fields of application, the adoption of hypertree decompositions in practice has been slow. One crucial challenge that is limiting their more widespread use is the computational difficulty of constructing good HDs. In general, finding an HD for a given hypergraph H and width at most k is NP-hard and W[1]-hard when parameterised by k [24] but is tractable when k is fixed, i.e., the problem is in XP in the terminology of parameterised complexity. In fact, a significantly stronger upper bound can be given. Finding an HD of fixed width is in the complexity class LogCFL [24] (contained in NC^2), and therefore in theory, highly parallelisable [8]. However, the theoretical parallelisability of the problem is demonstrated by construction of an appropriate Alternating Turing Machine [7], and no practical algorithm that allows for effective parallel computation of HDs is known. Here we, to the best of our knowledge, propose the first such algorithm.

Related Work. HD computation has received significant attention recently. This is witnessed, for instance, by the development of the large benchmark data set HyperBench [12], novel algorithmic approaches [11, 12, 26], and being the subject of a recent PACE competition [9]. Moreover, a number of new theoretical results [19, 20] have been presented, which have deepened our understanding of the problem. Still, the development of a parallel algorithm for hypertree decomposition remains a critical open question.

The two state-of-the-art approaches for computing HDs, `det-`k`-decomp` [27] and `HtdLEO` [31], both rely on techniques that are inherently unsuitable for parallelisation. `det-`k`-decomp` is heavily reliant on extensive caching and would therefore require excessive coordination between threads. In `HtdLEO`, the problem is encoded as an SMT instance and is therefore limited by the lack of parallelisation strategies for SMT solvers. While both algorithms perform well on current benchmarks, their lack of parallelisation ultimately limits them when it comes to solving large instances, i.e., finding HDs of large hypergraphs. This situation is especially disappointing as, on the one hand, single-core performance apparently does not suffice to solve larger instances and, on the other hand, the problem is in fact highly parallelisable in theory.

Interestingly, in Reference [26], a parallel algorithm `BalancedGo` is proposed for a slightly more general problem of computing **generalised hypertree decompositions (GHDs)** [25]. In HDs,

the so-called *special condition* enforces certain constraints on the parent/child nodes in the decomposition tree, and the tree must therefore be treated as rooted. Crucially, this constraint is no longer enforced in GHDs, and it is therefore also no longer necessary to consider the decomposition tree to be rooted. This additional degree of freedom is a key factor in the design of `BalancedGo`, where it ultimately allows for simple reassembly of individual decompositions of subproblems into a GHD of the full hypergraph. However, this freedom comes at a significant additional computational cost as the corresponding decision problem for computing GHDs is NP-hard even for constant width 2 [20, 25] (i.e., it is not even in XP in the parameterised setting). In practice, this leads to an additional exponential factor in the algorithms' complexity in contrast to the complexity of algorithms for computing HDs.

As a final note, we observe that the parallel computation of *tree decompositions* has been heavily studied (see, e.g., References [5, 6, 28]). Roughly speaking, these methods intuitively rely on first computing a tree decomposition with width bounded in terms of the width parameter k, but possibly higher than k. In a second step this decomposition is used to find a decomposition of lower width. There is no known analogue to either step for HDs. It is unclear whether techniques for the efficient computation of tree decompositions on graphs can help in the computation of hypertree width in general. Fundamentally, in the hypertree width setting, the cardinality of individual bags in a decomposition is no longer boundable in terms of the width parameter, whereas such a bound is typically key for treewidth techniques. Similarly, using efficient algorithms to compute the treewidth for typical graph encodings of hypergraphs is not helpful in our setting. For example, hyperedges become large cliques in the Gaifman graph, which therefore always has treewidth at least the size of the largest edge (minus 1) and thus is essentially unrelated to the hypertree width. In summary, current approaches either are not amenable to effective parallelisation or compute GHDs and therefore potentially cause exponential additional cost. The goal of this article is to bridge this gap and develop a parallel algorithm for computing hypertree decompositions.

Our Contributions. As argued above, this goal is not achievable by a straightforward extension of current approaches. The two principal algorithms for HDs are inherently unsuited for parallelisation while the parallel algorithm for GHDs fundamentally relies on the fact that GHDs are unrooted. We therefore develop a new theoretical machinery that will allow us to construct HDs in an arbitrary order instead of being limited to a strict top-down or bottom-up construction of the HD. This machinery then allows us to build on some of the ideas of `BalancedGo` while avoiding the complexity of GHDs. Experimental evaluation demonstrates that the resulting algorithm combines the best of both worlds by scaling effectively with an increase of parallel threads while avoiding the exponential overhead of GHD computation. Our main contributions are as follows:

— We develop a new theoretical framework of extended hypergraphs and their balanced separation, and we show that extended hypergraphs always have a balanced separator. To actually find such balanced separators, it is crucial to apply a novel approach that determines pairs of parent and child nodes of an HD (rather than a single node) at a time.
— Based on these new results we propose a novel algorithm, `log-k-decomp`, that searches for balanced separators at arbitrary positions in a potential HD. We argue that our algorithm is well suited for parallelisation; in particular, we prove a logarithmic upper bound on the recursion depth.
— We identify a number of further optimisations of our basic algorithm, and we incorporate them into a parallelised reference implementation of `log-k-decomp`.
— We identify the worst-case running time of `log-k-decomp` as $n^{O(k \log(n))}$, where k is the width parameter and n the input size. This means that `log-k-decomp` is, theoretically, not

optimal for a problem in the complexity class XP, which the problem of checking whether a hypergraph H has $hw \leq k$ falls under. However, we show in our experiments that our reference implementation of `log-k-decomp` is clearly competitive when compared to other systems that implement XP algorithms, such as `det-k-decomp`. Moreover, in our hybrid approach of combining the splitting into subproblems (whose size is guaranteed to be halved at each step) by `log-k-decomp` with the sequential `det-k-decomp`, we are actually back to a worst-case running time of $O(n^{f(k)})$.

— We compare the performance of `log-k-decomp` to `det-k-decomp` and `HtdLEO` through experiments over the HyperBench benchmark [12]. We observe that `log-k-decomp` outperforms the state of the art significantly. Furthermore, we experimentally verify the parallel scaling behaviour of `log-k-decomp`.

This article is a revised and expanded version of Reference [17]. For our revision, we have added Section 6, which provides an illustrative example to better explain how our presented algorithm `log-k-decomp` works by going through an example run of it. We also added Section 7, which details a number of optimisations and improvements to the base version `log-k-decomp`, and we prove that these optimisations still allow for a sound and complete algorithm. In addition to this, we have since implemented significant improvements to our proof-of-concept implementation and managed to solve more instances. We detail in the significantly expanded empirical evaluation (Section 8) our improvements to the original implementation and provide updated and expanded results and statistical analyses that show how our implementation fares against the state of the art in computing HDs when running against the standard HyperBench benchmark.

Structure. We formally introduce important concepts and notation in Section 2. The theoretical framework of extended hypergraphs and their balanced separation is established in Section 3. Building on this framework, we introduce the core ideas of the `log-k-decomp` algorithm and establish a logarithmic bound on its recursion depth in Section 4. The proof details of the correctness of the `log-k-decomp` algorithm are given in Section 5. The following Section 6 illustrates the algorithm further through a detailed example. We then discuss further ways to optimise the base algorithm in Section 7. The results of our empirical evaluation are presented in Section 8. We conclude with Section 9.

2 PRELIMINARIES

Conjunctive Queries, Constraint Satisfaction Problems, and hypergraphs. A *hypergraph* $H = (V(H), E(H))$ is a pair consisting of a set of vertices $V(H)$ and a set of non-empty (hyper)edges $E(H) \subseteq 2^{V(H)}$. We may assume w.l.o.g. that there are no isolated vertices, i.e., for each $v \in V(H)$, there is at least one edge $e \in E(H)$ with $v \in e$. We can thus identify a hypergraph H with its set of edges $E(H)$ with the understanding that $V(H) = \{v \in e \mid e \in E(H)\}$. A *subhypergraph* H' of H is then simply a subset of (the edges of) H. By slight abuse of notation, we may thus write $H' \subseteq H$ with the understanding that $E(H') \subseteq E(H)$ and, hence, implicitly also $V(H') \subseteq V(H)$. We are frequently dealing with sets of sets of vertices (e.g., sets of edges). For $S \subseteq 2^{V(H)}$, we write $\bigcup S$ as a short-hand for the union of such a set of sets, i.e., for $S = \{s_1, \ldots, s_\ell\}$, we have $\bigcup S = \bigcup_{i=1}^{\ell} s_i$. Figure 1 serves as an illustrative example of a hypergraph.

Conjunctive Queries (CQs) are arguably one of the most fundamental types of queries in the database world. Similarly, **Constraint Satisfaction Problems (CSPs)** are among the most fundamental formalisms in Artificial Intelligence and for modelling combinatorial problems. Formally, both are given by a first-order formula ϕ using only the connectives in $\{\exists, \wedge\}$ and disallowing $\{\forall, \vee, \neg\}$. Given such a formula ϕ, the hypergraph H_ϕ corresponding to ϕ is defined as follows: $V(H_\phi) = vars(\phi)$, i.e., the variables occurring in ϕ; and $E(H_\phi) = \{vars(a) \mid a \text{ is an atom in } \phi\}$.

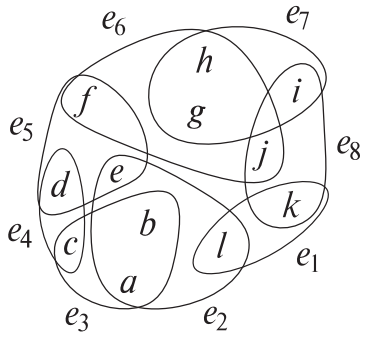


Fig. 1. A hypergraph consisting of 8 edges and 12 vertices.

In the sequel, we will only concentrate on hypergraphs, with the understanding that all results ultimately apply to CQs and CSPs.

Hypertree decompositions and hypertree width. We introduce the used notation first: Given a rooted tree $T = \langle N(T), E(T)\rangle$ with node set $N(T)$ and edge set $E(T)$, we write T_u to denote the subtree of T rooted at u, where u is a node in $N(T)$. Analogously, we write $T_u^{\uparrow}$ to denote the subtree of T induced by $N(T) \setminus N(T_u)$. Intuitively, T_u is the subtree of T "below" u and including u, while $T_u^{\uparrow}$ is the subtree of T "above" u. By slight abuse of notation, we sometimes write $u \in T$ instead of $u \in N(T)$ to denote that u is a node in T. Below, we shall introduce node-labelling functions χ and λ, which assign to each node $u \in T$ a set of vertices or edges, respectively, from some hypergraph H, i.e., $\chi(u) \subseteq V(H)$ and $\lambda(u) \subseteq E(H)$. For a node-labelling function f with $f \in \{\chi, \lambda\}$ and a subtree T' of T, we define $f(T')$ as $f(T') = \bigcup_{u' \in T'} f(u')$.

We are now ready to recall the definitions of hypertree decompositions and hypertree width from Reference [24]: An HD $\mathcal{D}$ of a hypergraph $H = (V(H), E(H))$ is a tuple $\mathcal{D} = \langle T, \chi, \lambda\rangle$, such that $T = \langle N(T), E(T)\rangle$ is a rooted tree, χ and λ are node-labelling functions with $\chi\colon N(T) \to 2^{V(H)}$ and $\lambda\colon N(T) \to 2^{E(H)}$, and the following conditions hold:

(1) for each $e \in E(H)$, there exists a node $u \in N(T)$ with $e \subseteq \chi(u)$;
(2) for each $v \in V(H)$, the set $\{u \in N(T) \mid v \in \chi(u)\}$ is connected in T;
(3) for each $u \in N(T)$, $\chi(u) \subseteq \bigcup \lambda(u)$;
(4) for each $u \in N(T)$, $\chi(T_u) \cap \left(\bigcup \lambda(u)\right) \subseteq \chi(u)$.

The *width* of an HD $\mathcal{D} = \langle T, \chi, \lambda\rangle$ is the maximum size of the λ-labels over all nodes $u \in T$, i.e., $width(\mathcal{D}) = \max_{u \in T} |\lambda(u)|$. Moreover, the *hypertree width* of a hypergraph H, denoted $hw(H)$, is the minimum width over all HDs of H. Condition (2) is called the "connectedness condition," and condition (4) is referred to as the "special condition" in Reference [24]. The set $\chi(u)$ is often referred to as the "bag" at node u, and we will also call it the "χ-label" of node u. Analogously, the set $\lambda(u)$ will be referred to as the "λ-label" of u.

If we drop the special condition from the above definition, then we get so-called GHD. The width of a GHD is again defined as the maximum size of the λ-labels over all nodes $u \in T$, and the *generalized hypertree width* of a hypergraph H, denoted $ghw(H)$, is the minimum width over all GHDs of H. The problem of checking if an HD of width $\leq k$ exists, and, if so, computing a concrete HD of width $\leq k$ is known to be feasible in polynomial time for arbitrarily chosen but fixed k [24]. In contrast, for GHDs, this problem has been shown to be NP-complete even if we fix $k = 2$ [20, 25]. Indeed, the special condition makes a huge difference when computing a decomposition. Intuitively, its effect is the following: When constructing a GHD or HD top-down, one "guesses" so to speak $\lambda(u)$ for each node u, starting from the root node. In case of

fixed k, there are only polynomially many choices for $\lambda(u)$. But then, we have to determine also $\chi(u)$. By Condition (3), $\chi(u)$ must be a subset of $\bigcup \lambda(u)$. However, without the special condition, effectively all subsets of $\bigcup \lambda(u)$ need to be considered, creating exponentially many choices for $\chi(u)$. Ultimately, this is the reason why GHD computation is NP-complete even for $k = 2$. In contrast, the special condition restricts the possible choices for $\chi(u)$ significantly, which makes HD computation tractable. Actually, as we will see in Section 3, it even allows us to define a normal form of HDs in which $\chi(u)$ is fully determined when we know $\lambda(u')$ of the parent u' of u and $\lambda(u)$. If u is the root, then the special condition simply implies $\chi(u) = \bigcup \lambda(u)$.

Applications of hypertree decompositions are often formulated in terms of GHDs. As noted above, if we drop the special condition from the definition of HDs, then we obtain precisely the definition of GHDs, and hence every HD is also a GHD. In principle, there are two differences between HDs and GHDs in applications. First, as already discussed, computing HDs is of lower computational complexity than computing GHDs. Second, due to the special condition restricting some possibilities, it is possible that $ghw(H) < hw(H)$ [25]. However, it is known that $hw(H)$ will never be much higher than $ghw(H)$, and indeed it holds that $hw(H) \leq 3\ ghw(H) + 1$ [2], i.e., hypertree width is guaranteed to be at most three times higher than generalized hypertree width. In practice, the situation is even better as there is no solved hypergraph in the standard benchmark Hyperbench [12] where $hw(H) \neq ghw(H)$ holds. Hence, for practical purposes, computing HDs can be seen as a more efficient way of computing GHDs.

Throughout this article, we will be dealing with a hypergraph H and a tree T of an HD of H. To avoid confusion, we will consequently refer to the elements in $V(H)$ as *vertices* (of the hypergraph) and to the elements in $N(T)$ as the *nodes* of T (of the decomposition).

3 CONNECTION SUBHYPERGRAPHS AND THEIR BALANCED SEPARATION

We introduce here an extension of subhypergraphs and then proceed to define the needed definitions of hypertree decomposition, components, and balanced separation on this new type of extended subhypergraph.

The key idea of our algorithm is to split the task of constructing an HD into subtasks of constructing *parts* of the HD, which will be referred to as "HD-fragments" in the sequel. These HD-fragments can later be stitched together to form an HD of a given hypergraph. This splitting into HD-fragments is realised by choosing a node u of the HD and splitting the HD into one subtree above node u and possibly several subtrees rooted at child nodes of u. The crux of our decomposition algorithm will be that the HD-fragment corresponding to the subtree $T_u^{\uparrow}$ above u and the HD-fragments corresponding to the subtrees T_{u_i} rooted at the child nodes u_i of u can be computed independently, that is, *in parallel*. However, at the end, these HD-fragments have to be stitched together to form a larger HD-fragment and, ultimately, to form the entire HD of the original hypergraph H. To this end, we have to keep track of, for each HD-fragment, how it connects to other HD-fragments.

The important interface information for the HD-fragment corresponding to the subtree $T_u^{\uparrow}$ above node u is the contents of $\chi(u)$. We thus introduce the notion of *special edges*. In case of the HD-fragment "above" node u (i.e., the HD-fragment corresponding to the subtree $T_u^{\uparrow}$) this special edge is simply the set $\chi(u)$ of vertices. Similarly, for each of the subtrees T_{u_i} rooted at the child nodes u_i of u, we have to keep track of the interface to $\chi(u)$ in the form of a set *Conn* of vertices, which is the intersection $\chi(T_{u_i}) \cap \chi(u)$.

3.1 Connection Subhypergraphs and Their HDs

At the heart of our decomposition algorithm in Section 4 will be a recursive function `Decomp`, which takes as input a subset E' of the edges $E(H)$, a set of special edges *Sp*, and a set of vertices *Conn*.

The goal of Decomp is to construct a fragment of an HD, such that every edge $e \in E'$ is covered by some node u' in the HD-fragment (i.e., $e \subseteq \chi(u')$), all special edges are covered by some leaf node of this HD-fragment (hence, these are the interfaces to the HD-fragments "below"), and *Conn* must be fully contained in $\chi(r)$ of the root r of this HD-fragment (hence, this is the interface to the HD-fragment "above"). Formally, the function Decomp deals with *extended subhypergraphs* of H in the following sense.

Definition 3.1 (Connection Subhypergraph). Let H be a hypergraph. An extended subhypergraph with connection interfaces (or *connection subhypergraph*, for short) of H is a triple $\langle E', Sp, Conn\rangle$ with the following properties:

— E' is a subset of the edge set $E(H)$ of H;
— Sp is a set of special edges, i.e., $Sp \subseteq 2^{V(H)}$;
— $Conn$ is a set of vertices, i.e., $Conn \subseteq V(H)$.

Notice that both edges and special edges are sets of vertices. Special edges are, in general, not edges of the hypergraph. They are generated in the course of our decomposition algorithm to keep track of the interface between HD-fragments that may then be constructed independently. To easily refer to the vertices of a connection subhypergraph $H' = \langle E', Sp, Conn\rangle$, we shall use the notation $V(H')$, where we define $V(H') = \bigcup E' \cup \bigcup Sp \cup Conn$.

We now extend several crucial definitions introduced in Reference [24] for hypergraphs to connection subhypergraphs. We shall also introduce a different type of normal form, one that will deviate in one crucial point from the one introduced in Reference [24].

Definition 3.2 (Hypertree Decomposition). Let H be a hypergraph, and let $H' = \langle E', Sp, Conn\rangle$ be a connection subhypergraph of H. An HD of H' is a tuple $\langle T, \chi, \lambda\rangle$, such that $T = \langle N(T), E(T)\rangle$ is a rooted tree, and χ and λ are node-labelling functions with $\lambda\colon N(T) \to 2^{E(H)\cup Sp}$ and $\chi\colon N(T) \to 2^{V(H')}$, such that the following conditions hold:

(1) for each $u \in N(T)$:
If $\lambda(u) = \{s\}$ for some $s \in Sp$, then u is a leaf in T and $\chi(u) = s$,
else $\lambda(u) \subseteq E(H)$ and $\chi(u) \subseteq \bigcup \lambda(u)$ must hold;
(2) each $f \in E' \cup Sp$ is "covered" by some $u \in N(T)$, i.e.:
If $f \in Sp$, then $\lambda(u) = \{f\}$ and, hence, $\chi(u) = f$,
else $f \in E'$ and $f \subseteq \chi(u)$ must hold;
(3) for each $v \in V(H')$, the set $\{u \in N(T) \mid v \in \chi(u)\}$ is connected in T;
(4) for each $u \in N(T)$, $\chi(T_u) \cap \left(\bigcup \lambda(u)\right) \subseteq \chi(u)$;
(5) the root r of T satisfies $Conn \subseteq \chi(r)$.

Clearly, H can also be considered as a connection subhypergraph of itself by taking the triple $\langle E(H), \emptyset, \emptyset\rangle$. Then the HDs of the connection subhypergraph $\langle E(H), \emptyset, \emptyset\rangle$ and the HDs of hypergraph H coincide.

The ultimate goal of the above definition of an HD of a connection subhypergraph of some hypergraph H is to capture fragments of an HD of a hypergraph H, which can be stitched together to actually yield an HD of the entire hypergraph H. This is also the reason why, in Condition (1) above, edges in $\lambda(u)$ may be chosen from the entire set $E(H)$ of edges in H. But ultimately, the "task" of an HD-fragment is to cover all edges and special edges of H'. That is why in Conditions (2) and (3) only E' (and not $E(H)$) is mentioned.

We note that we are slightly sloppy in using the terms "fragment of an HD" or, synonymously, "HD-fragment": In the first place, we thus mean a part of an HD of the original hypergraph H. The goal of our decomposition algorithm is to split the task of constructing an HD of H into pieces and to construct such HD-fragments. However, as mentioned above, we need to keep track of

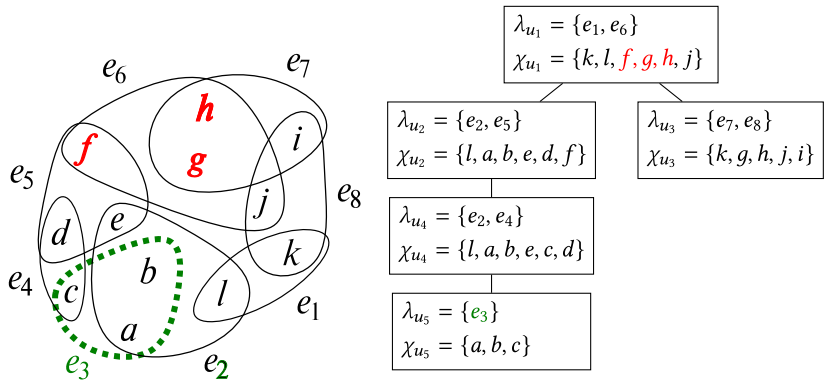


Fig. 2. An example connection subhypergraph of the hypergraph from Figure 1. The vertices are represented by letters. Special edges are marked by green and dotted hyperedges, and the vertices in Conn are marked in bold, red font. Next to it is an HD of width 2 of this connection subhypergraph.

the interfaces between various HD-fragments. In particular, special edges are introduced to keep track of the interface between an HD-fragment above some node u in the final HD and the HD-fragments below this node u. Therefore, we also refer to an HD that consists of a part of the final HD plus possibly some leaf nodes that cover special edges (i.e., that contain the aforementioned interface information) as an "HD-fragment." Actually, HDs of the connection subhypergraph that we consider in our decomposition algorithm are such HD-fragments, i.e., they contain a part of an HD of the original hypergraph H plus possibly leaf nodes that cover the special edges of H'.

Example 3.3. An example of a connection subhypergraph is shown in Figure 2, as well as an HD of this connection subhypergraph. We can see that no λ-label uses more than two hyperedges, and thus this HD has width 2, and the *hw* of the hypergraph is ≤ 2. In fact, the hypergraph contains alpha cycles [10], e.g., $\{e_2, e_3, e_4, e_5\}$. Hence, we also know that its *hw* must be > 1. Taken together, its *hw* is therefore exactly 2.

In Reference [24], Definition 5.1, a normal form of HDs was introduced. In Definition 3.8, we will carry the notion of normal form over to HDs of connection subhypergraphs. To this end, it is convenient to first define the set of (possibly special) edges *covered for the first time* (in top-down direction) by some node or by some subtree of an HD.

Definition 3.4. Let $H' = \langle E', Sp, Conn\rangle$ be a connection subhypergraph of some hypergraph H, and let $\mathcal{D} = \langle T, \chi, \lambda\rangle$ be an HD of H'. For a node $u \in T$, we write $cov(u)$ to denote the set of edges and special edges *covered for the first time* at u, i.e., $cov(u) = \{f \in E' \cup Sp \mid f \subseteq \chi(u)$ and for all ancestor nodes u' of u, $f \not\subseteq \chi(u')$ holds$\}$. For a subtree T' of T, we define $cov(T') = \bigcup_{u \in T'} cov(u)$.

Example 3.5. To illustrate the function *cov*, we shall give some example evaluations of it when applied to the HD given in Figure 2. Clearly, any (special) edge covered at the root node u_1 is covered for the first time in an HD, so we have $cov(u_1) = \{e_1, e_6\}$. For the internal node u_4, we have $cov(u_4) = \{e_4, e_3\}$. So we see that the special edge e_3 is actually covered for the first time in u_4. The node u_5 is only needed to satisfy Condition (2) of Definition 3.2, where we require that each special edge must appear in a node with a single edge in the edge cover, namely the special edge itself.

Components of a hypergraph or, more generally, of a connection subhypergraph, play a crucial role in the construction of a decomposition and also for the definition of our normal form for HDs of a connection subhypergraph.

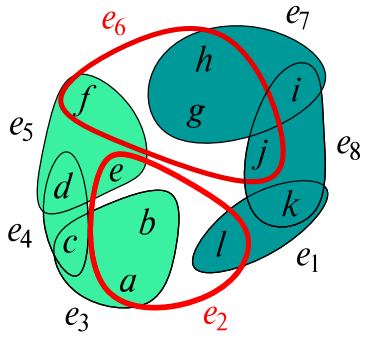


Fig. 3. Connected components and their respective separator, visually marked.

Definition 3.6 (Connectedness, Components). Let H be a hypergraph, let $U \subseteq V(H)$ be a set of vertices, and let $H' = \langle E', Sp, Conn \rangle$ be a connection subhypergraph of H.

— We define $[U]$-*adjacency* as a binary relation on $E' \cup Sp$ such that two (possibly special) edges $f_1, f_2 \in E' \cup Sp$ are $[U]$-*adjacent* if $(f_1 \cap f_2) \setminus U \neq \emptyset$ holds.
— We define $[U]$-*connectedness* as the transitive closure of the $[U]$-*adjacency* relation.
— A $[U]$-*component* of H' is a maximally $[U]$-connected subset $C \subseteq E' \cup Sp$.

Example 3.7. An example for a separator that generates multiple connected components can be seen in Figure 3. The separator S is the union of $e_2 \cup e_6$, marked via thicker edges. The corresponding $[S]$-components $C_1 = \{e_3, e_4, e_5\}$ and $C_2 = \{e_1, e_7, e_8\}$ are highlighted visually. Note that edges fully covered by S (in our example, e_2 and e_6) are not part of any $[S]$-component.

Let S be a set of edges and special edges with $U = \bigcup S$. Then we will also use the terms $[S]$-connectedness and $[S]$-components as a short-hand for $[U]$-connectedness and $[U]$-components, respectively.

3.2 Normal Form of HDs of Connection Subhypergraphs

Analogously to the normal form of HDs of hypergraphs in Reference [24], we now introduce a normal form of HDs of connection subhypergraphs.

Definition 3.8 (HD-normal Form of connection subhypergraphs). Let $H' = \langle E', Sp, Conn \rangle$ be a connection subhypergraph of some hypergraph H, and let $\mathcal{D} = \langle T, \chi, \lambda \rangle$ be an HD of H'. We say that $\mathcal{D}$ is in *normal form* if for every node p in T and every child node c of p the following properties hold:

(1) The set $cov(T_c)$ of edges is a $[\chi(p)]$-component of H';
(2) there exists $f \in cov(T_c)$ with $f \subseteq \chi(c)$;
(3) $\chi(c) = \big(\bigcup \lambda(c)\big) \cap \big(\bigcup cov(T_c)\big)$.

We will show in Claim A of the proof of Theorem 3.10 that, in any HD of H', $cov(T_c)$ is the union of *one or several* $[\chi(p)]$-components. Condition 1 of the normal form requires something stronger, namely that $cov(T_c)$ is exactly one $[\chi(p)]$-component C_p of H'.

Condition 2 serves to eliminate redundant nodes from an HD. As will be seen in the "Condition 2"-part of the proof of Theorem 3.10, if there is no such $f \in cov(T_c)$ with $f \subseteq \chi(c)$, then $\chi(c) \subseteq \chi(p)$, and we can actually delete node c from the HD.

Condition 3 is the only place where we deviate from the normal form in Reference [24]. The purpose of Condition 3 in Reference [24] is to make sure that $\chi(c)$ is uniquely determined whenever $\lambda(c)$, $\chi(p)$, and $cov(T_c)$ are known. However, there also would have been other choices to achieve

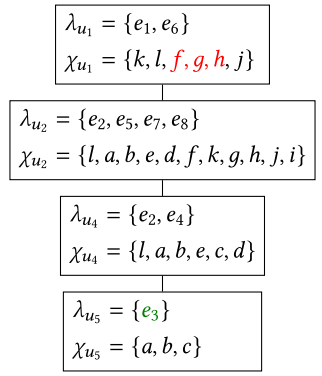


Fig. 4. An example for a non-normal form HD. This HD decomposes the connection subhypergraph given earlier in Figure 2.

this goal. Our Condition 3 chooses $\chi(c)$ *minimally*. That is, to ensure the special condition, $\chi(c)$ must contain all vertices from $\bigcup \lambda(c)$ that occur in $\chi(T_c)$. Since, by definition, all edges in $cov(T_c)$ are covered at some node in T_c, all vertices from $\big(\bigcup \lambda(c)\big) \cap \big(\bigcup cov(T_c)\big)$ must occur in $\chi(c)$. However, as will become clear in the proof of Theorem 3.10, there is no need to add further vertices to $\chi(c)$, since vertices not occurring in $\bigcup cov(T_c)$ can never violate the connectedness condition at node c as long as we stick to our strategy of choosing $\chi(u)$ minimally also for all nodes $u \in T_c$. In contrast, Condition 3 in Reference [24] chooses $\chi(c)$ *maximally*. That is, also all vertices in $\big(\bigcup \lambda(c)\big)$ that occur in $\chi(p)$ are added to $\chi(c)$. This deviation from the normal form in Reference [24] is crucial for the design of our algorithm, since, in our construction of an HD, we will be able to derive the possible sets $cov(T_c)$ as soon as we have $\lambda(p)$ and $\lambda(c)$, but we will "know" $\chi(p)$ only much later in the algorithm.

In the sequel, to improve readability, we will often use the name C_p when referring to the unique $[\chi(p)]$-component of H' from Condition 1 of Definition 3.8, which coincides with $cov(T_c)$.

Example 3.9. We already saw an example of a normal form HD in Figure 2 with the connection subhypergraph right next to it. We shall call this connection subhypergraph $H' = \langle E', Sp, Conn\rangle$, where $E' = \{e_1, e_2, e_4, e_5, e_6, e_7, e_8\}$, $Sp = \{e_3\}$, and $Conn = \{f, g, h\}$. This connection subhypergraph in turn is based on the hypergraph we saw earlier in Figure 1. To illustrate the differences between normal form and non-normal form HDs, we give an example of an HD that violates the normal form stated in Definition 3.8 in Figure 4. To understand why it does not satisfy Definition 3.8, we note that there are exactly two $[\chi_{u_1}]$-components of H', and yet u_1 has only one child node. Condition (1) requires for each $[\chi_{u_1}]$-component of H' to be covered in a separate subtree, whereas the non-normal form HD covers both components in the same subtree.

We now carry over two key results from Reference [24], whose proofs can be easily adapted to our setting of connection subhypergraphs.

Theorem 3.10 (cf. Reference [24], Theorem 5.4). *Let H' be a connection subhypergraph of some hypergraph H, and let $\mathcal{D}$ be an HD of H' of width k. Then there exists an HD $\mathcal{D}'$ of H' in normal form, such that $\mathcal{D}'$ also has width at most k.*

Proof. The proof proceeds in several steps:

Claim A (cf. Reference [24]. Lemma 5.2). *Let p, c be nodes in the HD $\mathcal{D}$ such that p is the parent of c. Moreover, let C' be a $[\chi(p)]$-component of H' with $C' \cap cov(T_c) \neq \emptyset$. Then $C' \subseteq cov(T_c)$ must hold.*

Proof of Claim A. We proceed by contradiction. Suppose that $C' \cap cov(T_c) \neq \emptyset$, but $C' \not\subseteq cov(T_c)$ holds. That is, there exist (possibly special) edges $e, e' \in C'$ such that $e' \in cov(T_c)$ and $e \notin cov(T_c)$, i.e., e is covered by some node $u \in N(T)$ outside T_c. However, e, e' are $[\chi(p)]$-connected, i.e., there exists a sequence of (possibly special) edges $e = e_1, e_2, \ldots, e_\ell = e'$ such that any two neighbouring edges e_i, e_{i+1} are $[\chi(p)]$-adjacent. By assumption, e_1 is covered by some node outside T_c, and e_ℓ is covered by some node in T_c. Hence, there must exist edges e_i, e_{i+1} such that e_i is covered by some node outside T_c and e_{i+1} is covered by some node in T_c. But e_i and e_{i+1} are $[\chi(p)]$-adjacent, i.e., they share a vertex x that is not in $\chi(p)$. So e_i is actually covered by a node outside T_c that is different from p. Hence, vertex x violates the connectedness condition of HDs (Condition 3 of Definition 3.2)—a contradiction. □

Claim B (cf. Reference [24]. Lemma 5.3). *Let p be a node in the HD $\mathcal{D}$. Moreover, let C be a $[\chi(p)]$-connected set of edges and special edges of H', and let $V = \left(\bigcup C\right) \setminus \chi(p)$. Moreover, for any set of vertices $X \subseteq V(H')$, let $nodes(X)$ denote the set of nodes u in the HD $\mathcal{D}$ with $\chi(u) \cap X \neq \emptyset$. Then $nodes(V)$ forms a connected subtree of $\mathcal{D}$.*

Proof of Claim B. We proceed by induction of the number of edges in C. First, consider a single edge e. By the connectedness condition of HDs (Condition 3 of Definition 3.2), we must have that, for every $x \in e \setminus \chi(p)$, $nodes(\{x\})$ induces a connected subtree of $\mathcal{D}$. Moreover, by Condition 2 of Definition 3.2, there exists a node u in $\mathcal{D}$ with $e \subseteq \chi(u)$. Hence, $u \in nodes(\{x\})$ for every $x \in e \setminus \chi(p)$, and, therefore, also $nodes(e \setminus \chi(p))$ forms a connected subtree of $\mathcal{D}$.

For the induction step, it suffices to show that, for any two (possibly special) edges e, e', if e, e' are $[\chi(p)]$-adjacent, then $nodes((e \cup e') \setminus \chi(p))$ forms a connected subtree of $\mathcal{D}$. We have just shown that $nodes(e \setminus \chi(p))$ and $nodes(e' \setminus \chi(p))$ form connected subtrees of $\mathcal{D}$. Moreover, since e, e' are $[\chi(p)]$-adjacent, they share a variable $x \notin \chi(p)$. By Condition 3 of Definition 3.2, $nodes(\{x\})$ is a connected subtree of $\mathcal{D}$. Hence, this latter subtree connects the two subtrees $nodes(e \setminus \chi(p))$ and $nodes(e' \setminus \chi(p))$ and, therefore, also $nodes((e \cup e') \setminus \chi(p))$ forms a connected subtree of $\mathcal{D}$. □

We are now ready to show that any HD can be transformed into an HD in normal form without increasing the width. This transformation proceeds top-down. Consider an arbitrary HD $\mathcal{D} = (T, \chi, \lambda)$ of a connection subhypergraph H'. Let (p, c) be a pair of nodes in $\mathcal{D}$ that violates one of the Conditions 1–3 of the normal form, and suppose that p is highest up in the tree T with this property. Then we can transform $\mathcal{D}$ as follows to ensure that (p, c) satisfies the Conditions 1–3.

Condition 1. By Claim A, we have that $cov(T_c)$ is the union of *one or several* $[\chi(p)]$-components. By Condition 1 of the normal form, there is *exactly one* $[\chi(p)]$-component C_p of H' satisfying $C_p \subseteq cov(T_c)$ and, therefore, $C_p = cov(T_c)$. Suppose that this is not the case, i.e., there exist $[\chi(p)]$-components $C_1, \ldots C_\ell$ with $C_i \subseteq cov(T_c)$ and $\ell \geq 2$. For every i, let $V_i = \left(\bigcup C_i\right) \setminus \chi(p)$. By Claim B, $nodes(V_i)$ forms a connected subtree of T_c. Let T_i denote this subtree. We take an isomorphic copy T_i' of T_i. For every node $u \in N(T_i)$, we write u' to denote the corresponding node in $N(T_i')$ under the isomorphism between T_i and T_i'. Then we define labelling functions χ_i' and λ_i' on the nodes in T_i' as follows: For every $u' \in N(T_i')$, we set $\chi_i'(u') = \chi(u) \cap \left(\bigcup C_i\right)$ and $\lambda_i'(u') = \lambda(u)$.

Now we transform $\mathcal{D}$ as follows: We delete the entire subtree T_c from T. Instead, we append the trees $T_1', \ldots, T_\ell'$ as new subtrees immediately below p, i.e., the roots $r_1', \ldots, r_\ell'$ of the trees $T_1', \ldots, T_\ell'$ become new child nodes of p. Clearly, every pair (p, r_i') satisfies Condition 1 of the normal form, i.e., there exists a $[\chi(p)]$-component C_p of H' such that $C_p = cov(T_c)$, namely $C_p = C_i$. Moreover, it is easy to verify that none of the Conditions 1–5 of Definition 3.2 is violated by this transformation.

Condition 2. Suppose that (p, c) is a pair of nodes in $\mathcal{D}$ that satisfies Condition 1 but violates Condition 2 of the normal form. Let C_p denote the $[\chi(p)]$-component with $cov(T_c) = C_p$. By the

violation of Condition 2, there is no $f \in C_p$ with $f \subseteq \chi(c)$. Note that then there is also no $f \in C_p$ with $f \in \lambda(c)$. Indeed, suppose to the contrary that there were some $f \in C_p$ with $f \in \lambda(c)$. Then, by the special condition of HDs (Condition 4 of Definition 3.2), all vertices in f would have to be in $\chi(c)$, and Condition 2 of the normal form would not be violated by (p, c).

It follows that $\lambda(c)$ contains only edges and special edges that are covered outside T_c. Hence, by the connectedness condition of HDs (Condition 3 of Definition 3.2), we must have $\chi(c) \subseteq \chi(p)$. But then we may simply delete the node c from T and turn all child nodes of c into child nodes of p. Clearly, this does not lead to a violation of any of the conditions of the definition of HDs.

Condition 3. Suppose that (p, c) is a pair of nodes in $\mathcal{D}$ that satisfies Condition 1 and 2 but violates Condition 3 of the normal form. Let C_p denote the $[\chi(p)]$-component with $cov(T_c) = C_p$. By Condition 1 of HDs in Definition 3.2, we have $\chi(c) \subseteq \left(\bigcup \lambda(c)\right)$. Moreover, by the connectedness condition of HDs (Condition 3 of Definition 3.2), we must have $\chi(p) \cap \left(\bigcup C_p\right) \subseteq \chi(c)$.

We can thus perform the following operation on the entire subtree T_c: All vertices from $\chi(T_c)$ that are outside $\left(\bigcup C_p\right)$ shall be removed from the χ-labels of the nodes in T_c. That is, for any $n \in T_c$, we may set $\chi(n) = \left(\bigcup \lambda(n)\right) \cap \left(\bigcup C_p\right)$. From the reasoning above, it follows that this may not lead to any violations of the conditions of Definition 3.2. □

3.3 Properties of Connection Subhypergraphs and Their Normal Form HDs

We move on to showing two key properties of connection subhypergraphs. First, we show that property (1) of Definition 3.2 can be strengthened. Not only is there always an HD such that $cov(T_c)$ is a $[\chi(p)]$-component, but in fact there is also an HD such that $cov(T_c)$ is always a $[\lambda(p)]$-component. Second, we show that a connection subhypergraph can always be separated in a balanced fashion by a bag of one of its HDs.

Toward the first goal we first show the following intermediate lemma.

Lemma 3.11 (cf. Reference[24], Lemma 5.8). *Let H' be a connection subhypergraph of some hypergraph H and let $\mathcal{D} = \langle T, \chi, \lambda \rangle$ be an HD in normal form of H'. Moreover, let p, c be nodes in T such that p is the parent of c and let $C_c \subseteq C_p$ for some $[\chi(p)]$-component C_p of H'. Then the following equivalence holds: C_c is a $[\chi(c)]$-component of H' if and only if C_c is a $[\lambda(c)]$-component of H'.*

Proof. It is convenient to prove the following claim first:

Claim A. *Let H' be a connection subhypergraph of some hypergraph H and let $\mathcal{D} = \langle T, \chi, \lambda \rangle$ be an HD in normal form of H'. Moreover, let p, c be nodes in T such that p is the parent of c and C_p is the $[\chi(p)]$-component of H' with $C_p = cov(T_c)$. Then, for any two (possibly special) edges $e, e' \in C_p$, the following equivalence holds: e and e' are $[\chi(c)]$-adjacent if and only if e and e' are $[\lambda(c)]$-adjacent.*

Proof of Claim A. By Condition 1 of Definition 3.2, we have $\chi(c) \subseteq \bigcup \lambda(c)$. Hence, $[\lambda(c)]$-adjacency clearly implies $[\chi(c)]$-adjacent. We only need to prove the opposite implication. So suppose $e, e' \in C_p$ are $[\chi(c)]$-adjacent. That is, they share some vertex x with $x \notin \chi(c)$. We show that e, e' are $[\lambda(c)]$-adjacent by proving that $x \notin \bigcup \lambda(c)$. By the *special condition* of HDs (Condition 4 of Definition 3.2), we have $\chi(T_c) \cap \left(\bigcup \lambda(c)\right) \subseteq \chi(c)$. Moreover, $C_p = cov(T_c)$ and, therefore, $\bigcup C_p \subseteq \chi(T_c)$. Taking these two subset-relationships together, we thus get $\bigcup C_p \cap \left(\bigcup \lambda(c)\right) \subseteq \chi(c)$. Therefore, if $x \in \bigcup \lambda(c)$ were the case, then, together with $x \in e' \in C_p$, we would have $x \in \chi(c)$. But x was chosen with the property $x \notin \chi(c)$. Hence, also $x \notin \bigcup \lambda(c)$ holds. □

Proof of the "if" direction. Suppose that C_c is a $[\chi(c)]$-component of H'. By Claim A, if two (possibly special) edges $e, e' \in C_p$ are $[\chi(c)]$-adjacent, then they are also $[\lambda(c)]$-adjacent. Hence, any $[\chi(c)]$-connected subset of C_p is also $[\lambda(c)]$-connected. Moreover, by $\chi(c) \subseteq \bigcup \lambda(c)$, a *maximal*

$[\chi(c)]$-connected set is also *maximal* $[\lambda(c)]$-connected. Hence, inside C_p, a $[\chi(c)]$-component is also a $[\lambda(c)]$-component.

Proof of the "only if" direction. Suppose that C_c is a $[\lambda(c)]$-component of H'. Hence, in particular, it is $[\lambda(c)]$-connected, and it is maximal with this property. Clearly, by $\chi(c) \subseteq \bigcup \lambda(c)$. C_c is also $[\chi(c)]$-connected. It remains to show that it is maximal with this property.

We proceed by contradiction. Suppose to the contrary that C_c is not maximally $[\chi(c)]$-connected. Then there exists a (possibly special) edge $e' \in C_p \setminus C_c$ such that e' is $[\chi(c)]$-adjacent to some $e \in C_c$. By Claim A, then e and e' are also $[\lambda(c)]$-adjacent. However, this contradicts the assumption that C_c is maximally $[\lambda(c)]$-connected. □

The principal proof ideas of Theorem 3.10 and Lemma 3.11 are the same as for the corresponding results (Theorem 5.4 and Lemma 5.8) in Reference [24]. The main technical difference between the proofs here and there comes from the fact that we have defined $[U]$-components in Definition 3.6 as sets of *edges and special edges* while they are defined as sets of *vertices* in Reference [24]. We have made this decision in order to simplify the presentation of our decomposition algorithm in the next section. In addition, some of the proof arguments for Theorem 3.10 and Lemma 3.11 could be slightly simplified compared with the corresponding results in Reference [24].

As has been noted above, our deviation from Reference [24] in the definition of the χ-label of nodes in a normal form HD is essential for the design of our decomposition algorithm to be presented in the next section. However, as far as the proof arguments of Theorem 3.10 and Lemma 3.11 are concerned, this deviation is inessential, since the "downward" components in an HD are not affected by adding or removing vertices from the χ-label of the parent node to the χ-label of the child node. However, for our purposes, we need a slightly stronger version of the above lemma: Recall that the HD construction in Reference [24] proceeds in a strict top-down fashion. Hence, when dealing with $\lambda(c)$, the bag $\chi(p)$ is already known. This is due to the fact that, initially at the root r, we have $\chi(r) = \bigcup \lambda(r)$ by the special condition. And then, whenever $\lambda(c)$ is determined and $\chi(p)$ plus a $[\chi(p)]$-component are already known, $\chi(c)$ can also be computed. However, in our HD algorithm, which "jumps into the middle" of the HD to be constructed, we do not automatically have $\chi(p)$ available when determining $\lambda(c)$. Hence, we need to slightly extend the above lemma to the following corollary, which allows us to identify $[\chi(p)]$-components with $[\lambda(p)]$-components also for the parent node p.

Corollary 3.12. *Let H' be a connection subhypergraph of some hypergraph H and let $\mathcal{D} = \langle T, \chi, \lambda \rangle$ be an HD in normal form of H'. Moreover, let p, c be nodes in T such that p is the parent of c. Then, the following property (1') holds: There exists a $[\lambda(p)]$-component C_p of H' such that $C_p = cov(T_c)$.*

Proof. We distinguish the two cases as to whether p is the root node of T or not.

Root node. Let $p = r$. Moreover, let c be a child node of r. We know, by the definition of the normal form, that $cov(T_c)$ is a $[\chi(r)]$-component. It remains to show that $cov(T_c)$ is also a $[\lambda(r)]$-component. To this end, we first show (analogously to the proof of Claim A in the proof of Lemma 3.11) that two edges $e, e' \in cov(T_c)$ are $[\chi(r)]$-adjacent if and only if they are $[\lambda(r)]$-adjacent.

By Condition 1 of Definition 3.2, we have $\chi(r) \subseteq \bigcup \lambda(r)$. Hence, $[\lambda(r)]$-adjacency clearly implies $[\chi(r)]$-adjacency. We only need to prove the opposite implication. So suppose $e, e' \in cov(T_c)$ are $[\chi(r)]$-adjacent. That is, they share some vertex x with $x \notin \chi(r)$. We show that e, e' are $[\lambda(r)]$-adjacent by proving that $x \notin \bigcup \lambda(r)$. Clearly, $x \in V(H') = \chi(T_r)$. Moreover, by the special condition of HDs (Condition 4 of Definition 3.2), we have $\chi(T_r) \cap \left(\bigcup \lambda(r)\right) \subseteq \chi(r)$. Hence, since we are assuming $x \notin \chi(r)$, we must have $x \notin \bigcup \lambda(r)$.

It is now easy to show that $cov(T_c)$ is a $[\lambda(r)]$-component: Since $cov(T_c)$ is a $[\chi(r)]$-component, we know, in particular, that $cov(T_c)$ is $[\chi(r)]$-connected. Hence, since we have just shown that $[\chi(r)]$-adjacency inside $cov(T_c)$ implies $[\lambda(r)]$-adjacency, we conclude that $cov(T_c)$ is also $[\lambda(r)]$-connected. Moreover, since $cov(T_c)$ is maximally $[\chi(r)]$-connected and $\chi(r) \subseteq \lambda(r)$ holds by the definition of HDs, $cov(T_c)$ is also maximally $[\lambda(r)]$-connected and, therefore, a $[\lambda(r)]$-component.

Node different from the root. Consider an arbitrary node p in T and let c be a child of p. Moreover, let $\hat{p}$ be the parent node of p. By the definition of the normal form, we know that $cov(T_p)$ is a $[\chi(\hat{p})]$-component $C_{\hat{p}}$ and $cov(T_c)$ is a $[\chi(p)]$-component C_p. Moreover, we have $C_p \subseteq C_{\hat{p}}$. By Lemma 3.11, we may conclude that C_p is a $[\lambda(p)]$-component. □

In Reference [12], balanced separators were used to design an algorithm for GHD computation. Below, we formally define balanced separators for our notion of connection subhypergraphs and we show that such a balanced separator always exists.

Definition 3.13 (Balanced Separators). Let H' be a connection subhypergraph of some hypergraph H, and let $\mathcal{D} = \langle T, \chi, \lambda \rangle$ be an HD of H'. A node u of T is a *balanced separator* if the following holds:

— for every subtree T_{u_i} rooted at a child node u_i of u, we have $|cov(T_{u_i})| \leq \frac{|E'|+|Sp|}{2}$ and
— $|cov(T_u^{\uparrow})| < \frac{|E'|+|Sp|}{2}$.

Intuitively, this means that none of the subtrees "below" u covers more than half of the edges of $E' \cup Sp$ and the subtree "above" u even covers less than half of the edges of $E' \cup Sp$.

Lemma 3.14. *Let H' be a connection subhypergraph of some hypergraph H, and let $\mathcal{D} = \langle T, \chi, \lambda \rangle$ be an HD of H'. Then there exists a balanced separator in $\mathcal{D}$.*

Proof. We show that, given an arbitrary HD, we can always find a balanced separator as follows: Initially, we set $u = r$ for the root node r of T and distinguish two cases: If $|cov(T_{u_i})| \leq \frac{|E'|+|Sp|}{2}$ holds for every subtree T_{u_i} rooted at a child node u_i of u, then u is a balanced separator and we are done. Otherwise, there exists a child node u_i of u such that $|cov(T_{u_i})| > \frac{|E'|+|Sp|}{2}$ holds for the subtree T_{u_i} rooted at u_i. Of course, there can exist only one such child node u_i. Moreover, by $cov(T_{u_i}^{\uparrow}) \cap cov(T_{u_i}) = \emptyset$, we have $|cov(T_{u_i}^{\uparrow})| < \frac{|E'|+|Sp|}{2}$.

Now set $u = u_i$ and repeat the case distinction: If $|cov(T_{u_i})| \leq \frac{|E'|+|Sp|}{2}$ holds for every subtree T_{u_i} rooted at a child node u_i of u, then u is a balanced separator and we are done. Otherwise, there exists a child node u_i of u such that $|cov(T_{u_i})| > \frac{|E'|+|Sp|}{2}$ holds for the subtree T_{u_i} rooted at u_i. Again, there can only be one such u_i. So we set $u = u_i$ and iterate the same considerations. This process is guaranteed to terminate, since, eventually, we will reach a leaf node of T. □

4 THE LOG-K-DECOMP ALGORITHM

Algorithm log-k-decomp, whose pseudo-code description is shown in Algorithm 1, aims at constructing an HD in normal form according to Definition 3.8 of width $\leq k$ for a given hypergraph H and integer $k \geq 1$. W.l.o.g., we assume that H is connected, since, otherwise, we can always compute an HD for each connected component of H separately and then construct an HD of H by combining the HDs of the connected components in an arbitrary way, e.g., choose the HD of one connected component and append the root node of any other HD as additional child of arbitrary nodes of the chosen HD. It is easy to verify that no condition of the definition of HDs can thus be violated.

The task of constructing an HD is split into subtasks that can then be processed in parallel. At the heart of log-k-decomp is the recursive function Decomp: It takes as input a connection

ALGORITHM 1: log-k-decomp

```
   Type: ConnSub=(E: Edge set, Sp: Special Edge set, Conn: Vertex set )
   Input: H: Connected Hypergraph
   Parameter: k: width parameter
   Output: true if hw of H ≤ k, else false
1  begin
2      H_comp := ConnSub(E: H, Sp: ∅, Conn: ∅)
3      return Decomp(H_comp)                                          ▷ initial call
4  function Decomp( H': ConnSub)
5      if |H'.E| ≤ k and |H'.Sp| = 0 then                             ▷ Base Cases
6          return true
7      else if |H'.E| = 0 and |H'.Sp| = 1 then
8          return true
9      foreach λ_p ⊆ H s.t. 0 ≤ |λ_p| ≤ k do                          ▷ ParentLoop
10         comps_p := [λ_p]-components of H'
11         if ∃i s.t. |comps_p[i]| > (|H'.E|+|H'.Sp|)/2 then
12             comp_down := comps_p[i]                                ▷ found child comp.
13         else
14             continue ParentLoop
15         if V(comp_down) ∩ H'.Conn ⊈ ⋃λ_p then
16             continue ParentLoop                                    ▷ connect. check
17         foreach λ_c ⊆ H s.t. 1 ≤ |λ_c| ≤ k do                       ▷ ChildLoop
18             χ_c := ⋃λ_c ∩ V(comp_down)
19             if V(comp_down) ∩ ⋃λ_p ⊈ χ_c then
20                 continue ChildLoop                                 ▷ connect. check
21             H_comp_down := ConnSub(E: comp_down.E, Sp: comp_down.Sp, Conn: ∅)
22             comps_c := [χ_c]-components of H_comp_down
23             if ∃i s.t. |comps_c[i]| > (|H'.E|+|H'.Sp|)/2 then
24                 continue ChildLoop
25             foreach x ∈ comps_c do
26                 Conn_x := V(x) ∩ χ_c
27                 H_x := ConnSub(E: x.E, Sp: x.Sp, Conn: Conn_x)
28                 if not(Decomp( H_x )) then
29                     continue ChildLoop                             ▷ reject child
30             comp_up.E := H'.E \ H_comp_down.E
31             comp_up.Sp := (H'.Sp \ H_comp_down.Sp) ∪ {χ_c}
32             H_comp_up := ConnSub(E: comp_up.E, Sp: comp_up.Sp, Conn: H'.Conn)
33             if not(Decomp(H_comp_up)) then
34                 continue ChildLoop                                 ▷ reject child
35             return true                                            ▷ hw of H' ≤ k
36     return false                                                   ▷ exhausted search space
```

subhypergraph H' of H in the form of parameter H' (with three fields $H'.E$, $H'.Sp$ and $H'.Conn$ for the sets of edges, special edges of H', and the interface of the HD-fragment to be constructed with the parts "above" in the final HD, respectively). This function returns "true" if an HD-fragment of width $\leq k$ of H' exists and "false" otherwise. The initial call to Decomp (line 3) is with the original hypergraph H viewed as connection subhypergraph with empty set of special edges and empty set

as connection interface. This connection subhypergraph is created by the call of the constructor ConnSub on line 2.

The key idea of function Decomp is to construct an HD of H' by finding a balanced separator (referred to as node c in Algorithm 1) of the HD to be constructed and to partition the set of edges and special edges of H' via this balanced separator. The HD construction then proceeds by recursively calling function Decomp for such subsets. More formally, let $C_1, \ldots, C_m$ denote all $[\chi(c)]$-components of H' and let $C_0 = \{e \in H'.E \cup H'.Sp \mid e \subseteq \chi(c)\}$. Then $H'.E \cup H'.Sp$ can be partitioned into the sets $C_0, \ldots, C_m$. For complexity reasons, our algorithm aims at finding an appropriate value of $\lambda(c)$ rather than $\chi(c)$. Clearly, for bounded width k of the desired HD, there are only polynomially many candidates for $\lambda(c)$ while there are, in general, exponentially many candidates for $\chi(c)$. By Corollary 3.12, for an HD in normal form, all $[\chi(c)]$-components of H' covered by a subtree rooted at a child of c are also $[\lambda(c)]$-components. But now we face two problems: (1) Which components of H' are covered by a subtree rooted at a child of c (as opposed to components that are covered by some node above the balanced separator c) and (2) how can we determine C_0, i.e., the edges and special edges covered by $\chi(c)$?

We solve these problems by also searching for the λ-label of the parent node p of c. Indeed, in an HD in normal form, $cov(T_c)$ corresponds to a $[\chi(p)]$-component C_p (referred to as $comp_{down}$ in Algorithm 1). By Corollary 3.12, C_p is also a $[\lambda(p)]$-component. This allows us to solve the first problem, since the $[\lambda(c)]$-components that are covered by a subtree rooted at a child of c are precisely those $[\lambda(c)]$-components that are a subset of C_p. Knowing $C_p = cov(T_c)$ also allows us to solve the second problem in a very simple way: We know that $cov(T_c)$ is partitioned into the (possibly special) edges covered by $\chi(c)$ and the (possibly special) edges contained in one of the $[\chi(c)]$-components inside C_p. Hence, C_0 consists of all (possibly special) edges in $cov(T_c)$ not contained in one of the $[\chi(c)]$-components inside C_p. In function Decomp, the set C_0 is only dealt with implicitly in that we call Decomp recursively for all $[\chi(c)]$-components inside C_p (on line 28) and for the connection subhypergraph containing all edges and special edges of H' outside C_p (on line 33). In other words, the edges and special edges of H' that are not contained in one of the $[\chi(c)]$-components inside C_p and that are outside C_p (that is, the edges and special edges covered by $\chi(c)$) are considered as done and are not contained in any recursive call to Decomp.

We now explain informally the various steps of function Decomp. A formal proof of the correctness of Algorithm 1 (in particular, of function Decomp) will be provided in Section 5, and an example will be worked out in detail in Section 6. The base case of function Decomp is reached (lines 5–8) when the existence of such an HD-fragment is trivial, i.e., either there are at most k edges and no special edges left or there is no edge and only one special edge left. In these cases, the desired HD-fragment simply consists of a single node whose λ-label either consists of the $\leq k$ edges or of the single special edge, respectively.

Function Decomp is controlled by two nested loops (lines 9–35 for the outer loop and lines 17–35 for the inner loop), which search for the λ-labels of two adjacent nodes p and c of the desired HD-fragment, such that p is the parent and c is the child. The idea of determining two nodes p and c has been explained above. In particular, we want node c to be a balanced separator of the connection subhypergraph H'. By Lemma 3.14, a balanced separator is guaranteed to exist. To find a balanced separator c, we have to make sure that node c satisfies the two conditions of Definition 3.13, i.e., (1) all of the subtrees rooted at a child of c cover at most half of the edges and special edges in H', and (2) the subtree $T_c^{\uparrow}$ "above" c covers strictly fewer than half of the edges and special edges in H'. For the second condition, observe that $comp_{down}$ (chosen at line 12) is meant to be covered precisely by T_c. Due to the fact that we are searching for an HD in normal form, we may assume that T_c covers exactly one $[\chi(p)]$-component $comp_{down}$, which, by Corollary 3.12, is also a $[\lambda(p)]$-component. Further observe that the edges and special edges covered by $T_c^{\uparrow}$ and the set $comp_{down}$

partition the edges and special edges in H'. Hence, checking if $comp_{down}$ contains more than half of H' (on line 11) is equivalent to checking condition (2), i.e., $T_c^{\uparrow}$ covers strictly fewer than half of the edges and special edges in H'. To check that c also satisfies the first condition of Definition 3.13, we have to compute all $[\lambda(c)]$-components inside $comp_{down}$ (line 22) and check that the size of each of them is at most half of the size of H' (line 23). Again, since we are only interested in HDs in normal form, we may assume here that each subtree rooted at a child of c covers exactly one of these $[\lambda(c)]$-components.

If such a balanced separator $\lambda(c)$ together with the λ-label $\lambda(p)$ at the parent node p of c has been found, then several checks have to be performed to make sure that the HD-fragment under construction satisfies the connectedness condition. For instance, all vertices in the intersection of *Conn* (i.e., the interface of the HD-fragment currently being constructed with the remaining HD "above" this HD-fragment) with component C_p (i.e., a component "below" node p) also have to occur in $\bigcup \lambda(p)$ (line 15).

Suppose that all these checks succeed. From $\lambda(p)$ and $\lambda(c)$, we can compute $\chi(c)$ according to Condition 3 of the normal form introduced in Definition 3.8 (line 18). In the HD $\mathcal{D}'$ to be constructed for the connection subhypergraph H', the edges and special edges of H' can be split into three disjoint categories as follows:

(1) the edges and special edges covered by $\chi(c)$,
(2) the edges and special edges covered by a subtree rooted at some child node of c, and
(3) the edges and special edges covered in the HD "above" c.

The edges and special edges covered by $\chi(c)$ are done and need no further consideration. The edges and special edges in the second and third category are taken care of by recursive calls to the function `Decomp` (lines 28 and 33). To this end, we compute all $[\chi(c)]$-components $C_1, \ldots, C_m$ (line 22). Now suppose that $C_1, \ldots, C_\ell$ with $1 \leq \ell \leq m$ are the $[\chi(c)]$-components inside the $[\lambda(p)]$-component C_p. Then the function `Decomp` is called recursively for each of the $[\chi(c)]$-components $C_1, \ldots C_\ell$ (line 28). In the call for component C_i, the interface $Conn_i$ is obtained simply as the intersection of the vertices in C_i and in $\chi(c)$ (line 26, where C_i is referred to as x). All of the remaining $[\chi(c)]$-components are taken care of by the HD-fragment "above" c, which we try to construct in another recursive call of function `Decomp` (line 33). In this recursive call, $\chi(c)$ is added as yet another special edge—in addition to the edges and special edges in the $[\chi(c)]$-components outside C_p. The additional special edge in the recursive call for the HD-part "above" node c and the interfaces *Conn* defined for each of the components as the intersection against $\chi(c)$, in the recursive calls for the HD-parts "below" node c ensure that we can (provided that all recursive calls of function `Decomp` are successful) stitch together the HD-fragments of these recursive calls to an HD-fragment of the connection subhypergraph H' of H.

To summarize, if all recursive calls return "true," then the overall result of this call to function `Decomp` is successful and returns "true" (line 35). If at least one of the recursive calls returns "false," then we have to search for a different label $\lambda(c)$ (in the next iteration of the "ChildLoop"). If eventually all candidates for $\lambda(c)$ have been tried out and none of them was successful, then we have to search for a different label $\lambda(p)$ of the parent node p (in the next iteration of the "ParentLoop") and restart the search for $\lambda(c)$ from scratch. Only when also all candidates for $\lambda(p)$ have been tried out and none of them was successful does function `Decomp` return the overall result "false" (line 36).

We conclude our informal discussion of Algorithm 1 by mentioning a slightly subtle point concerning the candidates for $\lambda(p)$ in the "ParentLoop." By the condition $0 \leq |\lambda_p| \leq k$ on line 9, $\lambda_p = \emptyset$ is also considered as a possible candidate. Of course, in the HD $\mathcal{D}'$ to be constructed for the connection subhypergraph H', we do not want to have nodes with empty λ-label and, hence, also empty χ-label. Nevertheless, this case is included for technical reasons. Actually, the check

on line 15 only succeeds if $H'.Conn$ is also empty. We are assuming that the input hypergraph H is connected. Hence, $H'.Conn = \emptyset$ can only occur when the HD $\mathcal{D}'$ to be constructed by the current call of Decomp contains the root of the HD $\mathcal{D}$ of H. In this case, p is a "dummy" parent of node c, and c will ultimately be the root of $\mathcal{D}'$ and, therefore, also of $\mathcal{D}$. Indeed, the effect of $\lambda_p = \emptyset$ is that $comps_p$ on line 10 contains a single $[\lambda_p]$-component, namely $H'.E \cup H'.Sp$. Consequently, the recursive call on line 33 for the component "above" node c is with a connection subhypergraph that contains no edge and only one special edge, namely $\chi(c)$. This means that the corresponding HD-fragment implicitly constructed for this call of Decomp consists of a single node u with $\lambda(u) = \{\chi(c)\}$ and $\chi(u) = \chi(c)$. When assembling the HD $\mathcal{D}'$ of H' from the HDs implicitly constructed by the various recursive calls of function Decomp, this node u gets merged with the root c of the rest of the HD $\mathcal{D}'$ of H' and, thus, disappears. The details of the construction of an HD from a successful call of function Decomp will be worked out in detail in the soundness proof in Section 5.

Below, we state the crucial property of log-k-decomp, which makes this approach particularly well suited for a parallel implementation.

Theorem 4.1. *Algorithm* log-k-decomp *correctly checks for given hypergraph H and integer $k \geq 1$ if $hw(H) \leq k$ holds. The algorithm is realised by a main program and the recursive function* Decomp*, whose recursion depth is bounded logarithmically in the number of edges of H, i.e., $O(\log(|H|))$. Moreover, by materialising the decompositions implicitly constructed in the recursive calls of the* Decomp *function, an HD of H of width $\leq k$ can be constructed in polynomial time in case of a successful computation (i.e., return-value "true").*

Proof. Theorem 4.1 has several parts. Below, we prove that the recursion depth is bounded logarithmically in the number of edges of H. The correctness of the algorithm log-k-decomp as well as the polynomial-time upper bound on the construction of an HD in case of a successful run of the algorithm (i.e., if it returns "true") will be proved in a separate section, namely Section 5.

The size of the connection subhypergraph in the call of function Decomp in the main program can only be bounded by the size of H itself. However, in every subsequent execution of Decomp for some connection subhypergraph $(H'.E, H'.Sp, H'.Conn)$, we always choose node c as a balanced separator. By Lemma 3.14, such a balanced separator always exists. The connection subhypergraphs in the recursive calls are therefore guaranteed to have size at most $\lceil \frac{|H'.E|+|H'.Sp|}{2} \rceil$. Note that the rounding up is necessary, because the new special edge $\chi(c)$ is added in the recursive call for the HD-fragment above c. Without this special edge, this component is guaranteed to be strictly smaller than $\frac{|H'.E|+|H'.Sp|}{2}$. At any rate, also with the upper bound $\lceil \frac{|H'.E|+|H'.Sp|}{2} \rceil$ on the size of the connection subhypergraphs of H in the recursive calls, we thus get an upper bound $O(\log(|H|))$ on the recursion depth. □

Note that we have formulated algorithm log-k-decomp as a decision procedure that decides if $hw(H) \leq k$ holds for given H and k. In case of a successful computation (i.e., return-value "true") it is easy to assemble a concrete HD of width $\leq k$ of H from the HD-fragments corresponding to the various calls of procedure Decomp. In Section 5, we shall discuss the construction of such a concrete HD as part of the correctness proof of Algorithm 1.

We emphasize two further important properties of algorithm log-k-decomp: First, it should be noted that the logarithmic bound on the recursion depth does not restrict the form of the HD in any way. In particular, it does not imply a logarithmic bound on the depth of the HD. The bound on the recursion depth is achieved by our novel approach of constructing the HD by recursively "jumping" to a balanced separator of the HD-fragment to be constructed rather than constructing the HD in a strict top-down manner as proposed in previous approaches [24, 27].

Second, as was explained at the beginning of this section, it is crucial in our approach that we search for appropriate λ-labels for *a pair* (p, c) *of nodes*, where p is the parent of c. One of the reasons why we need the λ-label of the parent is that this allows us to determine $\chi(c)$ from $\lambda(c)$. And only when we know $\chi(c)$ we can be sure which edges are indeed covered by $\chi(c)$. This knowledge is crucial to guarantee that all of the recursive calls of function `Decomp` have to deal with a connection subhypergraph whose size is halved, which in turn guarantees the logarithmic upper bound on the recursion depth. This strategy is significantly different from all previous approaches of decomposition algorithms. In References [12, 26], a parallel algorithm for generalised hypertree decompositions is presented. There, the problem of determining the χ-label of the balanced separator is solved by adding a big number of subedges to the hypergraph so that one may assume that $\chi(u) = \bigcup \lambda(u)$ holds for every node u. Clearly, this addition of subedges, in general, leads to a substantial increase of the hypergraph. In Reference [4], a preliminary attempt to parallelise the computation of HDs was made without handling pairs of nodes. However, in the absence of $\lambda(p)$, we cannot determine $\chi(c)$ from $\lambda(c)$. Consequently, we do not know which edges covered by $\bigcup \lambda(c)$ are ultimately covered by $\chi(c)$. Hence, all the edges covered by $\bigcup \lambda(c)$ would have to be added to the recursive call of `Decomp` for the HD-part "above" c, thus destroying the balancedness and the logarithmic upper bound on the recursion depth.

By Theorem 4.1, Algorithm `log-k-decomp` guarantees a logarithmic bound on the recursion depth and thus provides a good basis for a parallel implementation. Nevertheless, it still leaves room for several improvements. For instance, we can define also negative base cases to detect the overall answer "false" faster, we can restrict the edges that may possibly be used in the λ-labels of a connection subhypergraph (and provide them as an additional parameter of the function `Decomp`), and so on. These ideas and several further improvements—together with the pseudo-code of the resulting improved algorithm—are presented in Section 7.

The complexity of `log-k-decomp`. It is known that deciding whether $hw(H) \leq k$ holds for a given hypergraph H is in XP in terms of parameterised complexity, i.e., there is an $O(n^{f(k)})$ algorithm that checks whether a hypergraph with size n has width $\leq k$. However, our algorithm `log-k-decomp` does not achieve this bound and has a worst-case running time of only $n^{O(k \log n)}$: Each call of Decomp iterates through roughly n^{2k} separator pairs. We cannot significantly bound the number of balanced separators and thus each recursion incurs another multiplicative factor in the order of n^{2k} in the worst case. Since recursion can nest up to $\log n$ times, we observe the stated worst-case complexity.

The additional log factor in the exponent is an indirect consequence of our use of special edges. The canonical XP algorithm for deciding $hw \leq k$ [24] is stated in terms of a log-space **Alternating Turing Machine (ATM)** [7] (wrt. fixed k). It is known that log-space ATMs can be simulated by polynomial time deterministic Turing Machines [30]. However, this simulation relies on caching to avoid repetition of computation trees (cf. the `NewDetKDecomp` algorithm in Reference [12]). Notably, this cache is always of polynomial size due to the log-space bound on the representation of each ATM state. In our algorithm, keeping track of special edges is required to guarantee that decompositions for individual components fit together. Intuitively, this book-keeping of special edges makes the potential number of different computation trees exponential (in order $\log n$) and thus on a theoretical level, caching is no longer sufficient to achieve XP running time.

However, our empirical evaluation in Section 8 confirms that these concerns are mainly of theoretical nature as our algorithm clearly outperforms `NewDetKDecomp` from Reference [12], which is strictly in XP. In practice, we also employ a hybrid strategy, where we switch to the simpler `NewDetKDecomp` algorithm after a sufficient number of balanced splitting steps by `log-k-decomp`.

In Section 8, we will see that this performs best in practice as the benefit of balanced splitting diminishes once the connection subhypergraphs become small. Such a hybrid method in fact exhibits XP worst-case complexity if we switch to NewDetKDecomp at a constant depth d (leading to no more than $n^{O(kd)}$ calls of the $n^{O(k)}$ algorithm NewDetKDecomp). At any rate, in our implementation, we choose a more sophisticated hybridisation strategy (see Section 8.3) for better practical performance than the constant depth criterion.

log-k-decomp vs. BalancedGo. We conclude this section by pointing out the main difference between our new log-k-decomp algorithm for HD computation and the BalancedGo algorithm from Reference [26] for GHD computation. Both algorithms are based on "guessing" a balanced separator, computing components relative to this separator, and then recursively decomposing each component. However, as mentioned in Section 2, in case of GHDs, we do not have the special condition (Condition 4 in the definition of HDs recalled in Section 2) at our disposal to restrict the possible values of $\chi(u)$ when we know $\lambda(u)$. Hence, a different method for getting access to $\chi(u)$ and, in the next step, to the $[\chi(u)]$-components is required. Indeed, such a method was proposed in Reference [20] and then refined in Reference [26]: By adding certain subedges of the edges in $E(H)$, we may assume w.l.o.g. that $\chi(u) = \bigcup \lambda(u)$ holds. That is, the hypergraph $H = (V(H), E(H))$ is transformed into a new hypergraph $H' = (V(H), E'(H))$ such that, for every $e' \in E'(H)$, there exists $e \in E(H)$ with $e' \subseteq e$. In References [20, 26] a way to restrict the set $E'(H)$ is proposed so that we do not have to add all of $2^{E(H)}$. Nevertheless, in general, $E'(H) \setminus E(H)$ is still exponentially big. Of course, by the NP-hardness of GHD-computation even for fixed $k = 2$, this exponential blow-up is to be expected. This is in sharp contrast to our new log-k-decomp algorithm for HD-computation, where we never need to add any edges to the hypergraph. However, in log-k-decomp, we always have to "guess" $\lambda(p)$ of the parent p in addition to the balanced separator $\lambda(c)$, as was explained at the beginning of this section.

A minor technical difference comes from the fact that, for HDs, the definition of the root node is essential (due to the "special condition," i.e., Condition 4 of Definition 3.2), whereas a GHD can be rooted at any node. Therefore, in log-k-decomp, the interface of an HD-fragment to the neighbouring fragment above (i.e., the set *Conn* of vertices in our algorithm) and the interface to the neighbouring fragments below (i.e., the special edges) are treated differently. Consequently, in the splitting step of log-k-decomp, we add a special edge only to the recursive call for the component "above" node c. In contrast, BalancedGo treats the interfaces to all neighbouring GHD-fragments equally and, therefore, adds a special edge to each component resulting from a splitting step. Adding a special edge only to 1 recursive call in case of log-k-decomp (as opposed to adding a special edge to every recursive call in case of BalancedGo) further increases the complexity advantage of log-k-decomp over BalancedGo.

To conclude, the big advantage of log-k-decomp compared with BalancedGo is that log-k-decomp does not add any edges to the hypergraph while BalancedGo, in the worst case, has to add exponentially many new edges. The price to pay for this by log-k-decomp is that we always need to "guess" the λ-labels of pairs (p, c), where c is the balanced separator and p its parent. In BalancedGo, due to the added subedges, only $\lambda(c)$ needs to be "guessed." As will be seen in the experimental results in Section 8, the advantage of not having to add subedges clearly outweighs the disadvantage of having to deal with pairs (p, c) of nodes.

5 CORRECTNESS PROOF OF LOG-K-DECOMP

We prove the soundness and completeness of the algorithm log-k-decomp given in Algorithm 1 separately. The polynomial-time upper bound on the construction of an HD in case of a successful run of the algorithm (i.e., if it returns "true") will be part of the soundness proof.

It is convenient to first prove the following claim:

Claim A. *In every call of function* Decomp *in Algorithm 1 with parameter H' it is guaranteed that $H'.Conn \subseteq V(H')$ holds with $V(H') = (\bigcup H'.E) \cup (\bigcup H'.Sp)$.*

Proof of Claim A. The proof is by induction on the call depth of the recursive function Decomp.
induction begin. The top-level call of function Decomp on line 3 is using the connection subhypergraph H_{comp} as the parameter, where $H_{comp}.Conn$ is defined as $\emptyset$. Hence, $H_{comp}.Conn \subseteq V(H_{comp})$ trivially holds.
induction step. Suppose that Claim A holds for every call of function Decomp down to some call level n and suppose that function Decomp is called recursively during execution of Decomp at call level n. Suppose that this execution of Decomp is with parameter H'. The only places where function Decomp is called recursively are lines 28 and 33. More specifically, Decomp is called with parameter H_x on line 28 and with parameter $H_{comp_{up}}$ on line 33. We have to show that both $H_x.Conn \subseteq V(H_x)$ (on line 28) and $H_{comp_{up}}.Conn \subseteq V(H_{comp_{up}})$ (on line 33) hold. On line 28, the condition is trivially fulfilled, since we have that $H_x.Conn = Conn_x$, and $Conn_x$ is defined on line 26 as $Conn_x = V(x) \cap \chi(c)$.

It remains to consider the call of function Decomp on line 33. Suppose that $comps_c$ on line 22 is of the form $comps_c = \{x_1, \ldots, x_\ell\}$. By the definition of components in Definition 3.6, H' (that is, $H'.E \cup H'.Sp$) can be partitioned into the following disjoint subsets:

- $x_1.E \cup x_1.Sp, \ldots, x_\ell.E \cup x_\ell.Sp$
- $y = \{f \in H'.E \cup H'.Sp \mid f \subseteq \chi(c)\}$.
- $z = (H'.E \setminus comp_{down}.E) \cup (H'.Sp \setminus comp_{down}.Sp)$

We thus have $V(H') = \bigcup_{i=1}^{\ell} V(x_i) \cup V(y) \cup V(z)$ with $V(y) \subseteq \chi(c)$. By construction (line 22), all components x_i are contained in $comp_{down}$. Hence, we actually have $V(H') = V(comp_{down}) \cup \chi(c) \cup V(z)$. The recursive call of function Decomp on line 33 is with the edges and special edges in z plus $\chi(c)$ as an additional special edge. Hence, we have $V(comp_{up}) = V(z) \cup \chi(c)$ when Decomp is called on line 33 with parameter $H_{comp_{up}}$. It is therefore sufficient to show that $H'.Conn \subseteq V(z) \cup \chi(c)$ holds.

By the induction hypothesis, we may assume that $H'.Conn \subseteq V(H')$ holds. The check on line 15 ensures that $V(comp_{down}) \cap H'.Conn \cap \subseteq \bigcup \lambda(p)$ holds for the edge set $\lambda(p)$. Moreover, the check on line 19 ensures that $(\bigcup \lambda(p)) \cap V(comp_{down}) \subseteq \chi(c)$. In total, we thus have $H'.Conn \cap V(comp_{down}) \subseteq \chi(c)$. Together with $V(H') = V(comp_{down}) \cup \chi(c) \cup V(z)$ and $H'.Conn \subseteq V(H')$, we may thus conclude $H'.Conn \subseteq V(z) \cup \chi(c)$ and, therefore, $H'.Conn \subseteq V(comp_{up})$ (on line 33). Hence, also the call of function Decomp on line 33 satisfies Claim A. □

Soundness Proof. Suppose that algorithm log-k-decomp returns "true" when given some hypergraph H as input. We have to show that then H has an HD of width $\leq k$. Algorithm log-k-decomp returns "true" when the call of function Decomp on line 3 with the connection subhypergraph H_{comp} returns "true." Hence, it suffices to show that function Decomp is sound, i.e., if Decomp returns "true" when called with a connection subhypergraph H' as parameter, then H' has an HD of width $\leq k$. Moreover, we have to show that by materialising the decompositions implicitly constructed in the recursive calls of function Decomp, an HD of width $\leq k$ of H' can be constructed in polynomial time whenever Decomp returns "true." The proof is by induction on $|H'.E| + |H'.Sp|$.
induction begin. Suppose that $|H'.E| + |H'.Sp| = 1$ and that function Decomp returns "true." Hence, we either have $|H'.E| = 1$ and $|H'.Sp| = 0$ or we have $|H'.E| = 0$ and $|H'.Sp| = 1$. In either case, an HD of this connection subhypergraph can be obtained with a single node u by setting $\lambda(u) = \{f\}$ and $\chi(u) = f$, where f is the only (special) edge in $H'.E \cup H'.Sp$. This decomposition

clearly satisfies all conditions of an HD according to Definition 3.2, the only non-trivial part being Condition 5: We have to verify $H'.Conn \subseteq \chi(u)$. By Claim A above, we know that in every call of function Decomp, $H'.Conn$ is a subset of the vertices in $H'.E \cup H'.Sp$. Now in case $|H'.E|+|H'.Sp| = 1$ holds, we have $H'.E \cup H'.Sp = \{f\}$ for a single (special) edge f and, therefore, $\chi(u) = f = \left(\bigcup H'.E\right) \cup \left(\bigcup H'.Sp\right)$. Hence, we indeed have $H'.Conn \subseteq \chi(u)$.

induction step. Now suppose that $|H'.E| + |H'.Sp| > 1$ and that function Decomp returns "true." This means that one of the return-statements in lines 6, 8, or 35 is executed. Actually, line 8 can be excluded for $|H'.E| + |H'.Sp| > 1$. Now consider the remaining two lines 6 and 35. If the return-statement on line 6 is executed, then we have $|H'.E| \leq k$ and $|H'.Sp| = 0$. In this case, analogously to the induction begin, the desired HD consists of a single node u with $\lambda(u) = H'.E$ and $\chi(u) = \bigcup H'.E$. Again, all conditions of an HD according to Definition 3.2 are easy to verify; in particular, the proof argument for Condition 5 is the same as above.

It remains to consider the case that "true" is returned on line 35. This means that, for a particular value of $\lambda(p)$ (chosen on line 9) and of $\lambda(c)$ (chosen on line 17), all recursive calls of function Decomp (on lines 28 and 33) return "true." By the induction hypothesis, we may assume that for each of the connection subhypergraphs processed by these recursive calls of Decomp, an HD of width $\leq k$ exists. Note that we are making use of Claim A here in that we may assume that all recursive calls of Decomp are with properly defined connection subhypergraphs (in particular, the vertex set supplied as third component of the parameter is covered by the edges and special edges in the first two components of the parameter of each such call). Now look at these recursive calls: We are studying a call of function Decomp with parameter H', where H' is a connection subhypergraph consisting of a set $H'.E$ of edges, a set $H'.Sp$ of special edges and a set $H'.Conn$ of vertices. The current call of function Decomp apparently has chosen labels $\lambda(p)$ and $\lambda(c)$ for nodes p and c, such that all checks on lines 11, 15, 19, and 23, are successful in the sense that program execution continues with these values of $\lambda(p)$ and $\lambda(c)$. In particular, there exists a $[\lambda(p)]$-component $comp_{down}$ of H', satisfying the conditions $V(comp_{down}) \cap H'.Conn \subseteq \bigcup \lambda(p)$ (line 15) and $V(comp_{down}) \cap \bigcup \lambda(p) \subseteq \chi(c)$ (line 19).

Let $\{x_1, \ldots, x_\ell\}$ denote the set of $[\chi(c)]$-components of H' inside $comp_{down}$. Then $H'.E \cup H'.Sp$ (i.e., the set of edges and special edges in H') can be partitioned into the following disjoint subsets:

- $x_1.E \cup x_1.Sp, \ldots, x_\ell.E \cup x_\ell.Sp$
- $(H'.E \setminus comp_{down}.E) \cup (H'.Sp \setminus comp_{down}.Sp)$
- $\{f \in H'.E \cup H'.Sp \mid f \subseteq \chi(c)\}$.

From the first two kinds of sets of edges and special edges, the following connection subhypergraphs are constructed, for which function Decomp is then called recursively on lines 28 and 33:

- for each x_i consisting of a set of edges $x_i.E$ and special edges $x_i.Sp$, define $H_i = (x_i.E, x_i.Sp, x_i.Conn)$ with $x_i.Conn = V(x_i) \cap \chi(c)$;
- for $(H'.E \setminus comp_{down}.E) \cup (H'.Sp \setminus comp_{down}.Sp)$ define $H^\uparrow = (E^\uparrow, Sp^\uparrow, Conn^\uparrow)$ with $E^\uparrow = H'.E \setminus comp_{down}.E)$ and $Sp^\uparrow = (H'.Sp \setminus comp_{down}.Sp) \cup \{\chi(c)\}$ and $Conn^\uparrow = H'.Conn$.

By assumption, the recursive calls of Decomp for each of these connection subhypergraphs return the value "true." Thus, by the induction hypothesis, for each of these connection subhypergraphs, there exists an HD of width $\leq k$. From these HDs, we construct an HD of H' as follows:

- First take the HD of $H^\uparrow$. We shall refer to this HD as $\mathcal{D}^\uparrow$. Let r denote the root node of $\mathcal{D}^\uparrow$. By $Conn^\uparrow = H'.Conn$, we have $H'.Conn \subseteq \chi(r)$.
- Recall that $\chi(c)$ was added as a special edge to the connection subhypergraph $H^\uparrow$. Hence, by Definition 3.2, the HD $\mathcal{D}^\uparrow$ has a leaf node u with $\lambda(u) = \{\chi(c)\}$ and $\chi(u) = \chi(c)$. Now we replace node u in $\mathcal{D}^\uparrow$ by node c with $\lambda(c)$ and $\chi(c)$ according to the current execution of function Decomp. Moreover, for every $f \in H'.Sp$ with $f \subseteq \chi(c)$, we append a fresh

child node c_f to c with $\lambda(c_f) = \{f\}$ and $\chi(c_f) = f$. It is easy to verify that the resulting decomposition (let us call it $\mathcal{D}'$) is an HD of the connection subhypergraph that contains all edges and special edges of H' except for the ones in any of the x_i's. In particular, node r with $H'.Conn \subseteq \chi(r)$ is still the root of HD $\mathcal{D}'$.

— Now we take the HDs $\mathcal{D}_i$ of the connection subhypergraphs $(x_i.E, x_i.Sp, x_i.Conn)$ and append them as subtrees below c in $\mathcal{D}'$, i.e., the root nodes of the HDs $\mathcal{D}_i$ become child nodes of c. Let us refer to the resulting decomposition as $\mathcal{D}$. It remains to show that $\mathcal{D}$ indeed is an HD of width $\leq k$ of the connection subhypergraph H' of H. The width is clear, since all HD-fragments of $\mathcal{D}$ and also $\lambda(c)$ have width $\leq k$. It is also easy to verify that every edge in $H'.E$ is covered by some node in $\mathcal{D}$ and every special edge in $H'.Sp$ is covered by some leaf node in $\mathcal{D}$. Moreover, also the connectedness condition holds inside each HD-fragment (by the induction hypothesis) and between the various HD-fragments. The latter condition is ensured by the definition of components in Definition 3.6 and by the fact that any two connection subhypergraphs processed by the various recursive calls of function Decomp can only share vertices from $\chi(c)$.

Finally, note that the above construction of HD $\mathcal{D}$ from the HD-fragments constructed in the recursive calls of Decomp is clearly feasible in polynomial time. □

Before we prove the completeness of algorithm log-k-decomp, we introduce a special kind of connection subhypergraphs: Let H be a hypergraph, and let $\mathcal{D} = \langle T, \chi, \lambda \rangle$ be an HD of H with root r. We call $H' = (E', Sp, Conn)$ a *$\mathcal{D}$-induced connection subhypergraph* of H if there exists a subtree T' of T with the following properties:

— $E' = cov(T')$;
— let B denote those nodes in T that are outside T' but whose parent node is in T'; then $Sp = \{\chi(u) \mid u \in B\}$.
— If the root r of T' has the parent p in T, then $Conn = V(H') \cap \bigcup \lambda(p)$. Otherwise, if the root r of T' is also the root of T, then $Conn = \emptyset$.

An HD $\mathcal{D}' = \langle S', \chi', \lambda' \rangle$ of H' is then obtained as follows:

— the tree S' of $\mathcal{D}'$ is the subtree of T induced by the nodes of T' plus the nodes in B;
— for all nodes u in T', we set $\chi'(u) = \chi(u)$ and $\lambda'(u) = \lambda(u)$;
— for all nodes u in B, we set $\chi'(u) = \chi(u)$ and $\lambda'(u) = \{\chi(u)\}$.

We shall refer to $\mathcal{D}'$ as the induced HD of H'. It is easy to verify that $\mathcal{D}'$ is in normal form, whenever $\mathcal{D}$ is in normal form. In the completeness proof below, we shall refer to a $\mathcal{D}$-induced connection subhypergraph of H simply as an "induced subhypergraph" of H. No confusion can arise from this, since we will always consider the same HD $\mathcal{D}$ of H throughout the proof.

Completeness Proof. Suppose that hypergraph H has an HD of width $\leq k$. We have to show that then algorithm log-k-decomp returns "true." This is the case if the call of function Decomp with parameter H_{comp} on line 3 returns "true," where H_{comp} is the connection subhypergraph $(E(H), \emptyset, \emptyset)$. We are assuming that H has an HD of width $\leq k$. Of course, this is then also an HD of H_{comp}. By Theorem 3.10, H_{comp} also has an HD $\mathcal{D} = \langle T, \chi, \lambda \rangle$ of width $\leq k$ in normal form. Moreover, H_{comp} is clearly a $\mathcal{D}$-induced connection subhypergraph of H. Hence, it suffices to show that function Decomp is complete on induced subhypergraphs of H. That is, if Decomp is called with an induced subhypergraph H' that has an HD of width $\leq k$, then Decomp returns "true." We proceed by induction on $|H'.E| + |H'.Sp|$.

induction begin. Suppose that $|H'.E|+|H'.Sp| = 1$. That is, we either have $|H'.E| = 1$ and $|H'.Sp| = 0$ or we have $|H'.E| = 0$ and $|H'.Sp| = 1$. In the first case, "true" is returned via the statement on line 6; in the second case, "true" is returned via the statement on line 8.

induction step. Now suppose that $|H'.E| + |H'.Sp| > 1$. If $|H'.E| \leq k$ and $|H'.Sp| = 0$, then the return-statement on line 6 is executed and the function returns "true." It remains to consider the case that $|H'.E| > k$ or $|H'.Sp| > 1$ holds. By Lemma 3.14, the HD $\mathcal{D}'$ induced by H' has a balanced separator. Let us refer to this balanced separator as the node c in $\mathcal{D}'$. By the balancedness, it can be easily verified that c must satisfy $\lambda(c) \subseteq E(H)$ (that is, c is not a leaf node with $\lambda(c) = \{f\}$ for some special edge f). We distinguish the following three cases.

Case 1. Suppose that c is the root node of $\mathcal{D}'$ and c is different from the root of $\mathcal{D}$. Hence, c has a parent node in $\mathcal{D}$. Let us refer to this parent node as p. If function Decomp has not already returned "true" before, then it will eventually try $\lambda(p)$ in the foreach-statement on line 9. Due to the normal form of $\mathcal{D}'$, all of H' is a single $[\lambda(p)]$-component. Hence, the if-condition on line 11 is satisfied and $comp_{down}$ is assigned all of H' on line 12. The connectedness check on line 15 succeeds, since p is the parent of the root of $\mathcal{D}'$ and $H'.Conn = V(H') \cap \bigcup \lambda(p)$ holds by the last condition of the definition of induced subhypergraphs. Hence, the foreach-loop on lines 17–35 is eventually entered. If function Decomp does not return "true" before, then it will eventually try $\lambda(c)$ in the foreach-statement on line 17. Then $\chi(c)$ assigned on line 18 is the correct χ-label of c according to the normal form. The connectedness check on line 19 succeeds, since $\mathcal{D}$ satisfies the connectedness condition. By assumption, c is a balanced separator; hence also the check on line 23 succeeds. Thus, the foreach-loop on lines 25–29 is executed. It is easy to verify that the parameters supplied to Decomp in the recursive calls on line 28 correspond to induced subhypergraphs. Therefore, all these calls of Decomp return "true" by the induction hypothesis. Hence, also the statements on lines 30–33 are executed. In this case, since $comp_{down}$ comprises all edges and special edges of H', Decomp is called on line 33 with $comp_{up}.E = \emptyset$ and $comp_{up}.Sp = \{\chi(c)\}$. Hence, as was shown in the induction begin, this call of Decomp returns "true." Therefore, the return-statement on line 35 is executed and the overall result "true" is returned by function Decomp.

Case 2. Suppose that c is the root node of $\mathcal{D}'$ and c is also the root of $\mathcal{D}$. It is easy to verify that, in this case, the call of Decomp with parameter H' satisfies $H'.Conn = \emptyset$. Indeed, the original call of Decomp on line 3 is with $H'.Conn = \emptyset$. Moreover, all recursive calls of Decomp on line 28 for the HD-fragment "above" node c let the *Conn*-part (i.e., the third part) of the parameter unchanged.

If function Decomp does not return "true" before, then it will eventually try $\lambda_p = \emptyset$ in the foreach-statement on line 9. Then $comps_p$ on line 10 contains a single $[\lambda_p]$-component, consisting of all edges and special edges in $H'.E \cup H'.Sp$. Hence, the check on line 11 succeeds and this single component will be assigned to $comp_{down}$ on line 12. By the above considerations, we have $H'.Conn = \emptyset$. Hence, the connectedness check on line 15 trivially succeeds and the foreach-loop on lines 17—35 is eventually entered. If function Decomp does not return "true" before, then it will eventually try $\lambda(c)$ in the foreach-statement on line 17. Then $\chi(c)$ is assigned the correct χ-label of c according to the normal form on line 18. Finally, the connectedness check on line 19 trivially succeeds due to $\lambda_c = \emptyset$.

From now on, the argumentation is exactly as in Case 1: By assumption, c is a balanced separator; hence also the check on line 23 succeeds. Thus, the foreach-loop on lines 25–29 is executed. It is easy to verify that the parameters supplied to Decomp in the recursive calls on line 28 correspond to induced subhypergraphs. Therefore, all these calls of Decomp return "true" by the induction hypothesis. Hence, also the statements on lines 30–33 are executed. In this case, since $comp_{down}$ comprises all edges and special edges of H', Decomp is called on line 33 with $comp_{up}.E = \emptyset$ and $comp_{up}.Sp = \{\chi(c)\}$. Hence, as was shown in the induction begin, this call of Decomp returns "true." Therefore, the return-statement on line 35 is executed, and the overall result "true" is returned by function Decomp.

Case 3. Suppose that c is not the root node of $\mathcal{D}'$. Then c has a parent node inside $\mathcal{D}'$. Let us refer to this parent node as p. If function Decomp has not already returned "true" before, then it will eventually try $\lambda(p)$ in the foreach-statement on line 9. By Corollary 3.12, the $[\chi(p)]$-component C_p consisting of the edges and special edges that are covered in $\mathcal{D}$ by the subtree rooted at child node c of p is also a $[\lambda(p)]$-component. Hence, one of the $[\lambda(p)]$-components computed on line 10 is this $[\chi(p)]$-component C_p. The check on line 11 is successful, because the child c of p is a balanced separator. Hence c and the subtrees below c cover more than half of the edges and special edges. The check on line 15 succeeds because of the connectedness condition in $\mathcal{D}$. Hence, the foreach-loop on lines 17–35 is eventually entered. If function Decomp does not return "true" before, then it will eventually try $\lambda(c)$ in the foreach-statement on line 17. Then $\chi(c)$ assigned on line 18 is the correct χ-label of c according to the normal form. The connectedness check on line 19 succeeds, since $\mathcal{D}$ satisfies the connectedness condition. By assumption, c is a balanced separator; hence also the check on line 23 succeeds. Thus, the foreach-loop on lines 25–29 is executed. It is easy to verify that the parameters supplied to Decomp in the recursive calls on line 28 correspond to induced subhypergraphs. Hence, all these calls of Decomp return "true" by the induction hypothesis. Hence, also the statements on lines 30–33 are executed. Again, it is easy to verify that also the parameters supplied to Decomp in the recursive call on line 33 correspond to an induced subhypergraph. Hence, by the induction hypothesis, also this call of Decomp returns "true." Therefore, the return-statement on line 35 is executed and the overall result "true" is returned by function Decomp. □

6 AN ILLUSTRATIVE EXAMPLE

To illustrate the notions introduced in Section 3 and the basic algorithm log-k-decomp shown in Algorithm 1, we consider the hypergraph $H = (V, E)$ with $V = \{x_1, \ldots, x_{10}\}$ and

$$E = \{R_1(x_1, x_2),\ R_2(x_2, x_3),\ R_3(x_3, x_4),\ R_4(x_4, x_5),\ R_5(x_5, x_6),\\ R_6(x_6, x_7),\ R_7(x_7, x_8),\ R_8(x_8, x_9), R_9(x_9, x_{10}),\ R_{10}(x_{10}, x_1)\ \}.$$

In other words, H is a the cycle graph with 10 vertices $x_1, \ldots, x_{10}$. A hypertree decomposition $\mathcal{D}$ of H is shown in Figure 5(a).

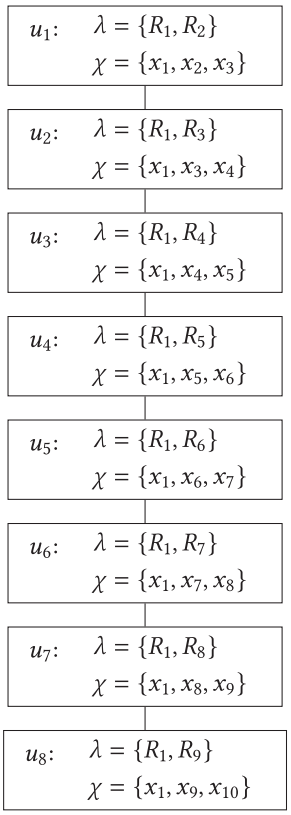
We now walk through algorithm log-k-decomp, which will allow us to see also the notions from Section 3 in action. Suppose that we run log-k-decomp with hypergraph H and parameter $k = 2$.

In each of the loops on lines 9 and 17, the algorithm searches for a λ-label until it finds a successful one. By "successful," we mean that the current execution of the main program or of function Decomp returns "true" (on lines 6, 8, or 35, respectively). To keep things simple in our discussion below, we will directly choose a successful one with the understanding that this particular λ-label will eventually be selected by the program unless another successful one has already been found before.

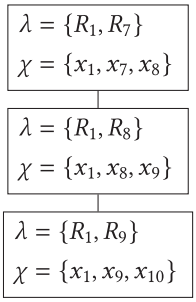
Main program. The algorithm begins by creating a connection subhypergraph H_{comp} with all the edges $\{R_1, \ldots, R_{10}\}$, and empty sets for the special edges and for *Conn*. Then we call the function Decomp with H_{comp} as input (line 3).

Call 1 of function Decomp with parameters $H'.E = \{R_1, \ldots, R_{10}\}$, $H'.Sp = \emptyset$, and $H'.Conn = \emptyset$. Since none of the conditions of the base case is satisfied, the ParentLoop will be entered. It will eventually try $\lambda_p = \{R_1, R_5\}$. This splits H' into 2 components $c_1 = \{R_2, R_3, R_4\}$ and $c_2 = \{R_6, \ldots, R_{10}\}$. Component c_2 satisfies the size constraint on line 11 and, therefore, becomes $comp_{down}$ on line 12.

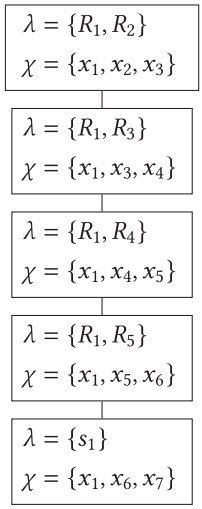
The ChildLoop will eventually try $\lambda_c = \{R_1, R_6\}$. On line 18, we thus set $\chi_c = \{x_1, x_6, x_7\}$. This gives rise to a single component $comps_c[i]$ inside $comp_{down}$, namely $comps_c[i] = \{R_7, R_8, R_9, R_{10}\}$.



(a) HD $\mathcal{D}$ of hypergraph H from Section 6



(b) HD-fragment $\mathcal{D}_{1.1}$ implicitly constructed by Call 1.1 of function Decomp



(c) HD-fragment $\mathcal{D}_{1.2}$ implicitly constructed by Call 1.2 of function Decomp

Fig. 5. Visualisations of the HD constructed as part of Section 6 and the HD-fragments used for its construction.

This component satisfies both the constraint for connectedness (line 19) and the size constraint (line 23). Moreover, it leads to two recursive calls of function Decomp: one for the component $comps_c[i] = \{R_7, R_8, R_9, R_{10}\}$ "below" the "child node" (on line 28) and one for the component $c_1 = \{R_1, R_2, R_3, R_4, R_5\}$ "above" (on line 33). For the component "below," we have $x.E = \{R_7, R_8, R_9, R_{10}\}$ and $x.Sp = \emptyset$; moreover, we set $x.Conn = \{x_1, x_7\}$ on line 26. For the component "above," we set $H_{comp_{up}}.E = \{R_1, R_2, R_3, R_4, R_5\}$ on line 30 and $H_{comp_{up}}.Sp = \{s_1\}$ with $s_1 = \chi_c = \{x_1, x_6, x_7\}$ on line 31; moreover, we retain $H_{comp_{up}}.Conn = \emptyset$ from the current call of Decomp. Clearly, edge R_6 from H' is already covered by $\chi_c = \{x_1, x_6, x_7\}$ and does not need to be further considered.

As we shall work out next, Call 1.1 of function Decomp for the component "below" will return "true" based on the HD-fragment $\mathcal{D}_{1.1}$ shown in Figure 5(b). Likewise, Call 1.2 of function Decomp for the component "above" will return "true" based on the HD-fragment $\mathcal{D}_{1.2}$ shown in Figure 5(c). The leaf node of $\mathcal{D}_{1.2}$ contains the special edge s_1, which acts as a placeholder for the node c with labels $\lambda_c = \{R_1, R_6\}$ and $\chi_c = \{x_1, x_6, x_7\}$ from the current call of Decomp.

The HD-fragment $\mathcal{D}_1$ of the successful Call 1 of function Decomp is then obtained by taking HD-fragment $\mathcal{D}_{1.2}$, replacing the leaf node with λ-label $\{s_1\}$ by the node c with $\lambda_c = \{R_1, R_6\}$ and $\chi_c = \{x_1, x_6, x_7\}$ and appending the HD-fragment $\mathcal{D}_{1.1}$ below this node c. In other words, the HD-fragment $\mathcal{D}_1$ is precisely HD $\mathcal{D}$.

Call 1.1 of function Decomp with parameters $H'.E = \{R_7, R_8, R_9, R_{10}\}$, $H'.Sp = \emptyset$, and $H'.Conn = \{x_1, x_7\}$. The ParentLoop (line 9) and the ChildLoop (line 17) will eventually choose $\lambda_p = \{R_1, R_7\}$ and $\lambda_c = \{R_1, R_8\}$, respectively. Then function Decomp is called recursively with parameters $x.E = \{R_9, R_{10}\}$, $x.Sp = \emptyset$, and $x.Conn = \{x_1, x_9\}$ on line 28 for the only component "below" and with parameters $H_{comp_{up}}.E = \{R_7\}$, $H_{comp_{up}}.Sp = \{s_2\}$ with $s_2 = \{x_1, x_8, x_9\}$, and $H_{comp_{up}}.Conn = \{x_1, x_7\}$ on line 33 for the component "above."

Call 1.1.1 of function Decomp with parameters $H'.E = \{R_9, R_{10}\}$, $H'.Sp = \emptyset$, and $H'.Conn = \{x_1, x_9\}$. This call immediately returns "true", since we have reached the base case on lines 5 and 6. The corresponding HD-fragment $\mathcal{D}_{1.1.1}$ consists of a single node with λ-label $\{R_9, R_{10}\}$.

Call 1.1.2 of function Decomp with parameters $H'.E = \{R_7\}$, $H'.Sp = \{s_2\}$ with $s_2 = \{x_1, x_8, x_9\}$, and $H'.Conn = \{x_1, x_7\}$. In the ParentLoop, eventually, $\lambda_p = \{R_1, R_6\}$ will be chosen on line 9. In this case, $comp_{down}$ is actually all of H', and it clearly satisfies the size constraint on line 11. In the ChildLoop, $\lambda_c = \{R_1, R_7\}$ will eventually be chosen on line 17. It gives rise to $\chi_c = \{x_1, x_7, x_8\}$ on line 18 with a single $[\chi_c]$-component $comps_c[i] = \{s_2\}$. Function Decomp will therefore be called on line 28 with parameters $H_x.E = \emptyset$, $H_x.Sp = \{s_2\}$, and $H_x.Conn = \{x_1, x_8\}$. This call (referred to as Call 1.1.2.1) returns "true", since we now have the base case on lines 7 and 8.

The recursive call of function Decomp on line 33 for the "components above" is a special case where there are actually no such "components above" left. Hence, in this case, we have $H_{comp_{up}}.E = \emptyset$, $H_{comp_{up}}.Sp = \{s_3\}$ with $s_3 = \chi_c = \{x_1, x_7, x_8\}$, and $H_{comp_{up}}.Conn = \{x_1, x_7\}$. This leads to the Call 1.1.2.2 of function Decomp, which returns "true", since we again have the base case on lines 7 and 8.

In total, Call 1.1.2 of function Decomp is successful and the corresponding HD-fragment $\mathcal{D}_{1.1.2}$ consists of two nodes: the node with λ-label $\{R_1, R_7\}$ and its child node with λ-label $\{s_2\}$

We can now also construct the HD-fragment $\mathcal{D}_{1.1}$ of the successful Call 1.1 of function Decomp. More precisely, HD-fragment $\mathcal{D}_{1.1}$ is obtained by taking HD-fragment $\mathcal{D}_{1.1.2}$, replacing the leaf node with λ-label $\{s_2\}$ by the node c with $\lambda_c = \{R_1, R_8\}$ and $\chi_c = \{x_1, x_8, x_9\}$ from Call 1.1 and appending the HD-fragment $\mathcal{D}_{1.1.1}$ below this node c. The resulting HD-fragment $\mathcal{D}_{1.1}$ is shown in Figure 5(b). That is, $\mathcal{D}_{1.1}$ is the subtree consisting of the bottom 3 nodes u_6, u_7, and u_8, of the final HD $\mathcal{D}$ displayed in Figure 5(a).

Call 1.2 of function Decomp with parameters $H'.E = \{R_1, R_2, R_3, R_4, R_5\}$, $H'.Sp = \{s_1\}$ with $s_1 = \{x_1, x_6, x_7\}$, and $H'.Conn = \emptyset$. The execution of this function call is very similar to the calls discussed above. Below, we therefore do not discuss in detail the remaining recursive calls inside Call 1.2. Instead, we only list for each such call the parameters, the balanced separator χ_c, and the corresponding HD-fragments.

In Call 1.2 of function Decomp, eventually the balanced separator with $\lambda_c = \{R_1, R_3\}$ and $\chi_c = \{x_1, x_3, x_4\}$ will be chosen, which gives rise to the recursive Calls 1.2.1 (line 28) and 1.2.2 (line 33) of function Decomp, which we briefly discuss below.

Call 1.2.1 of function Decomp with parameters $H'.E = \{R_4, R_5\}$, $H'.Sp = \{s_1\}$ with $s_1 = \{x_1, x_6, x_7\}$, and $H'.Conn = \{x_1, x_4\}$. In Call 1.2.1 of function Decomp, eventually the balanced separator with $\lambda_c = \{R_1, R_5\}$ and $\chi_c = \{x_1, x_5, x_6\}$ will be chosen, which in turn gives rise to the recursive Calls 1.2.1.1 (line 28) and 1.2.1.2 (line 33) of function Decomp. Both of these calls are successful (as we will show below) and the corresponding HD-fragment $\mathcal{D}_{1.2.1}$ will combine the HD-fragments from Calls 1.2.1.1 and 1.2.1.2: From Call 1.2.1.2 we will have a leaf-node with special edge s_5, which has the same vertices as χ_c. We replace this leaf with a new node with λ set to λ_c. Below this new node, we attach the resulting HD-fragment $\mathcal{D}_{1.2.1.1}$ from the corresponding call, which consists of only a single node with λ-label $\{s_1\}$. Below we briefly discuss how the two Calls 1.2.1.1 and 1.2.1.2 are processed.

Call 1.2.1.1 of function Decomp with parameters $H'.E = \emptyset$, $H'.Sp = \{s_1\}$ with $s_1 = \{x_1, x_6, x_7\}$, and $H'.Conn = \{x_1, x_6\}$. We now have a connection subhypergraph of H consisting of only a single special edge. Thus we hit the base case on line 8 and return "true". The corresponding HD-fragment $\mathcal{D}_{1.2.1.1}$ consists of one node: the node with λ-label $\{s_1\}$.

Call 1.2.1.2 of function Decomp with parameters $H'.E = \{R_4\}$, $H'.Sp = \{s_5\}$ with $s_5 = \{x_1, x_5, x_6\}$, and $H'.Conn = \{x_1, x_4\}$. As in the Call 1.1.2, we now have a connection subhypergraph of H consisting of a single edge and a single special edge. Analogously to Call 1.1.2, also Call 1.2.1.2 returns "true" and the corresponding HD-fragment $\mathcal{D}_{1.2.1.2}$ consists of 2 nodes: the node with λ-label $\{R_1, R_4\}$ and its child node with λ-label $\{s_5\}$.

Call 1.2.2 of function Decomp with parameters $H'.E = \{R_1, R_2\}$, $H'.Sp = \{s_4\}$ with $s_4 = \{x_1, x_3, x_4\}$, and $H'.Conn = \emptyset$. Analogously to the Call 1.2.1, we have a connection subhypergraph of H consisting of two edges and a single special edge, and again this call of Decomp will return "true". Since the recursive calls of Call 1.2.2 are nearly identical to Call 1.2.1, we omit mentioning them explicitly. The corresponding HD-fragment $\mathcal{D}_{1.2.2}$ consists of two nodes: the root node with λ-label $\{R_1, R_2\}$ and its child node with λ-label $\{s_4\}$. Note that the root of this fragment also happens to be the root of $\mathcal{D}$ itself, as shown in Figure 5(a). The reason for this lies in the fact that Call 1.2.2. is reached by always following the recursive call in line 33, where we determine the HD-fragment for the component above the child. As these HD-fragments make up the HD we are trying to construct, we are thus traversing this HD upwards and thus eventually reach the root node of $\mathcal{D}$.

We can now construct the HD-fragment $\mathcal{D}_{1.2}$ of the successful Call 1.2 by taking HD-fragment $\mathcal{D}_{1.2.2}$, replacing the leaf node with λ-label $\{s_4\}$ by the node c with $\lambda_c = \{R_1, R_3\}$ and $\chi_c = \{x_1, x_3, x_4\}$ from Call 1.2 and appending the HD-fragment $\mathcal{D}_{1.2.1}$ below this node c. The resulting HD-fragment $\mathcal{D}_{1.2}$ is shown in Figure 5(c).

7 FURTHER COMBINATORIAL OBSERVATIONS AND OPTIMISATIONS

As shown in Theorem 4.1, algorithm log-k-decomp introduced in Section 4 reaches the primary goal of splitting the HD-construction into subtasks with guaranteed upper bound on the size of the subtasks. In theory, this is enough to support parallelism. However, this basic algorithm still leaves a lot of room for further improvements. In this section, we present several optimisations, which are crucial to achieve good performance in practice. The line numbers below refer to Algorithm 1. However, in Algorithm 2, we will ultimately also give the pseudo-code for the enhanced algorithm where all the optimisations mentioned below are included.

Extension of the base case. The recursive function Decomp starts (on lines 5–8) with some simple checks that immediately give a "true" answer. In contrast, a "false" answer is only obtained in case of unsuccessful execution of the entire procedure. We could add the following negative case to the top of the procedure: If $H'.E = \emptyset$, then $|H'.Sp| \leq 1$ must hold. The rationale of this condition is that if there are no more edges in $H'.E$, then we would have to use only "old" edges (i.e., edges covered already at some node further up in the HD) in the λ-label to separate the remaining special edges. However, a λ-label consisting of "old" edges only is not allowed, since this would violate the second condition of the normal form in Definition 3.8.

Root of the HD-fragment. In the current form of procedure Decomp, we always "guess" a pair (p, c) of nodes, such that p is the parent of c. This also covers the case that c is the root node of the HD-fragment for the current connection subhypergraph. In this case, the parent node p would actually be the node immediately above this HD-fragment (in other words, p was the node from which the current call of Decomp happened) or a "dummy" node (if the balanced separator is the

ALGORITHM 2: Optimised log-k-decomp

Type: ConnSub=(E: Edge set, Sp: Special Edge set, $Conn$: Vertex set) ▷ Connection Subhypergraph
Input: H: Connected Hypergraph
Parameter: k: width parameter
Output: **true** if hw of $H \leq k$, else **false**

begin
  $H_{comp} \coloneqq$ ConnSub(E: H, Sp: $\emptyset$, $Conn$: $\emptyset$)
  return Decomp(H_{comp}) ▷ initial call
function Decomp(H': ConnSub, A: *Edge set*)
  if $|H'.E| \leq k$ **and** $|H'.Sp| = 0$ **then** ▷ **Base Cases**
    return true
  else if $|H'.E| = 0$ **and** $|H'.Sp| = 1$ **then**
    return true
  else if $|H'.E| = 0$ **and** $|H'.Sp| > 1$ **then**
    return false
  foreach $\lambda_c \subseteq A$ *s.t.* $\lambda_c \cap H'.E \neq \emptyset$ ***and*** $1 \leq |\lambda_c| \leq k$ **do** ▷ **ChildLoop**
    $comps_c \coloneqq [\lambda_c]$-components of H'
    if $\exists i$ *s.t.* $|comps_c[i]| > \frac{|H'.E|+|H'.Sp|}{2}$ **then**
      continue ChildLoop
    else if $H'.Conn \subseteq \bigcup \lambda_c$ **then** ▷ check if λ_c is root
      $\chi_c \coloneqq \bigcup \lambda_c \cap V(H')$
      foreach $y \in comps_c$ **do**
        $Conn_y \coloneqq V(y) \cap \chi_c$
        $H_y \coloneqq$ ConnSub(E: $y.E$, Sp: $y.Sp$, $Conn$: $Conn_y$)
        if ***not***(Decomp(H_y, A)) **then**
          goto ParentLoop
      return true ▷ c is root of H'
    foreach $\lambda_p \subseteq A$ *s.t.* $\lambda_p \cap H'.E \neq \emptyset$ ***and*** $1 \leq |\lambda_p| \leq k$ **do** ▷ **ParentLoop**
      $comps_p \coloneqq [\lambda_p]$-components of H'
      if $\exists i$ *s.t.* $|comps_p[i]| > \frac{|H'.E|+|H'.Sp|}{2}$ **then**
        $comp_{down} \coloneqq comps_p[i]$ ▷ found child comp.
      else
        continue ParentLoop
      $\chi_c \coloneqq \bigcup \lambda_c \cap V(comp_{down})$
      if $V(comp_{down}) \cap Conn \not\subseteq \bigcup \lambda_p$ **then**
        continue ParentLoop ▷ connect. check
      if $V(comp_{down}) \cap \bigcup \lambda_p \not\subseteq \chi_c$ **then**
        continue ParentLoop ▷ connect. check
      $H_{comp_{down}} \coloneqq$ ConnSub(E: $comp_{down}.E$, Sp: $comp_{down}.Sp$, $Conn$: $\emptyset$)
      $new\_comps_c \coloneqq [\chi_c]$-components of $H_{comp_{down}}$
      foreach $x \in new\_comps_c$ **do**
        $Conn_x \coloneqq V(x) \cap \chi_c$
        $H_x \coloneqq$ ConnSub(E: $x.E$, Sp: $x.Sp$, $Conn$: $Conn_x$)
        if ***not***(Decomp(H_x, A)) **then**
          continue ParentLoop ▷ reject parent
      $comp_{up}.E \coloneqq H'.E \setminus H_{comp_{down}}.E$
      $comp_{up}.Sp \coloneqq (H'.Sp \setminus H_{comp_{down}}.Sp) \cup \{\chi_c\}$
      $H_{comp_{up}} \coloneqq$ ConnSub(E: $comp_{up}.E$, Sp: $comp_{up}.Sp$, $Conn$: $H'.Conn$)
      $A_{up} \coloneqq A \setminus H_{comp_{down}}.E$ ▷ reducing A
      if ***not***(Decomp($H_{comp_{up}}$, A_{up})) **then**
        continue ParentLoop ▷ reject parent
      return true ▷ hw of $H' \leq k$
  return false ▷ exhausted search space

root of the HD of the input hypergraph H and, therefore, does not have a parent node). However, it would be more efficient to consider the case of "guessing" the root node of this HD-fragment explicitly. More precisely, we would thus first check for the label λ_p guessed in Decomp on line 9 (which, in the current version of the algorithm, is automatically treated as the "parent") if all $[\lambda_p]$-components have at most half the size of the current connection subhypergraph.

If this is the case, then we may use this node as the root of the HD-fragment to cover the current connection subhypergraph. This makes sense, since it corresponds precisely to the "search" for a balanced separator in the proof of Lemma 3.14. That is, if the root of the HD gives rise to components that are all at most half the size, then this may actually be the desired balanced separator.

If this is not the case, then we simply proceed with procedure Decomp in its present form, i.e, there exists exactly one $[\lambda_p]$-component whose size is bigger than half. So we take the guessed node as the parent and search for a balanced separator as a child of p in the direction of this oversized $[\lambda_p]$-component.

Allowed edges. The main task of procedure Decomp is to compute labels (i.e., edge sets) λ_p and λ_c of nodes p, c, which will ultimately be in a parent-child relationship in the HD. For these labels, Algorithm 1 imposes no restriction. That is, in principle, we would try all possible sets of $\leq k$ edges for these labels. However, not all edges actually make sense. We should thus add one more parameter to procedure Decomp indicating the edges that are allowed in a λ-label of the HD-fragment for this connection subhypergraph.

More specifically, in our search for the λ-label of some node u, we may exclude from the HD of the connection subhypergraph $H_{comp_{up}}$ (i.e., in the recursive call of function Decomp on line 33) all edges which are part of some component "below" u. The rationale of this restriction is that, by the special condition, using a "new" edge in a λ-label forces us to add all its vertices to the χ-label, i.e., it is fully covered in such a node. But then it cannot be part of a component whose edges are covered for the first time further down in the tree.

Note that we can yet further restrict the search for the label λ_c by requiring that at least one edge must be from $H'.E$, since choosing only "old" edges would violate the second condition of the normal form. As far as the label λ_p is concerned, the same kind of restriction can be applied if we first implement the previous optimisation of handling the root node of the current HD-fragment separately. If we indeed have to guess the labels λ_p and λ_c of *two* nodes p and c (i.e., the label λ_p guessed first was not a balanced separator), then both nodes p and c are *inside* the current HD-fragment. Hence, also the label λ_p must contain at least one "new" edge.

Searching for child nodes first. In Algorithm 1, we first look for λ-labels of potential parent nodes, and consider afterwards the λ-labels of potential child nodes. Only then do we check if the χ-label of the child is a balanced separator of the current subcomponent. We have observed that in many hypergraphs of HyperBench, balanced separators are rare, in the sense that only a small part of the search space will ever fulfil the properties required. Therefore we should first look for a potential child s.t. its λ-label is a balanced separator, and only afterwards try to find a fitting parent. While this may seem slightly unintuitive, it allows us to quickly detect cases where no balanced separator can be found at all.

In principle, we can determine the precise bag χ_c for a child c only when we know the λ-label of its parent. Nevertheless, even if we only have λ_c, we can over-approximate the χ_c-label as $\bigcup \lambda_c$. Hence, if $\bigcup \lambda_c$ is not a balanced separator, then we may clearly conclude that neither is χ_c. Actually, the "over-approximation" is not significant anyway: When we later determine an appropriate λ-label of the parent p of c, we will compute the "downward"-component C_p whose

size is bigger than $\frac{|H'.E|+|H'.Sp|}{2}$. The HD-fragment rooted at c is then supposed to cover all of C_p. By Corollary 3.12, we know that the "downward"-component C_p is both, a $[\lambda(p)]$-component and a $[\chi(p)]$-component. Hence, by Lemma 3.11, we know that, inside C_p, there is no difference between the $[\chi(c)]$-components and the $[\lambda(c)]$-components.

Finally, note that by searching for the child node first, we get the above described optimisation of treating the "Root of the HD-fragment" separately almost for free. Indeed, when computing λ_c, we can immediately check if $H'.Conn \subseteq \bigcup \lambda_c$ holds. Recall that $H'.Conn$ constitutes the interface to the HD-fragment *above* the current one. Hence, if $\bigcup \lambda_c$ fully covers this interface, then c may be the root node of the current HD-fragment, which is checked on lines 15–22 in Algorithm 2. Of course, even if λ_c is a balanced separator, this does not necessarily mean that it is indeed the root of the current HD-fragment. Hence, either the check on line 15 in Algorithm 2 or one of the recursive calls on line 20 may fail. In this case, we would simply proceed to the search for the label λ_p of the parent node p of c. Note that we can now be sure that p is inside the HD-fragment of H'. Hence, in contrast to Algorithm 1, the case of $\lambda_p = \emptyset$ does not need to be considered here, i.e., on line 23 in Algorithm 2, we have the condition $1 \leq |\lambda_p| \leq k$ rather than $0 \leq |\lambda_p| \leq k$ as was the case for the "ParentLoop" in Algorithm 1.

Speeding up the search for parent λ-labels. The previous optimisation means that, after having found a λ-label λ_c for the child that is a balanced separator of the current subcomponent, we need to find a *suitable* λ-label of the parent. By "suitable" we mean that we may limit ourselves to edges that have a non-empty intersection width $\bigcup \lambda_c$. A very high-level explanation why we may exclude edges e with $e \cap \bigcup \lambda_c = \emptyset$ from the search space of λ_p is that the control flow of function `Decomp` is mainly determined by the edges and special edges covered by $\bigcup \lambda_c$ and the $[\lambda_c]$-components *below* c. By the connectedness condition, if e is covered *above* c and has empty intersection width $\bigcup \lambda_c$, then excluding or including e in λ_p has no effect on the $[\lambda_c]$-components *below* c. In our experimental evaluation, we found that this restriction indeed significantly reduces the time it takes to either find a *suitable* λ_p, or detect that no such λ-label exists. Of course, this restriction of the search space cannot destroy soundness. We will show below that also the completeness of the algorithm is preserved.

Theorem 7.1. *The optimised* `log-k-decomp` *algorithm for checking if a hypergraph H has $hw(H) \leq k$ given in Algorithm 2 is sound and complete. More specifically, for given hypergraph H and integer $k \geq 1$, the algorithm returns "true" if and only if there exists an HD of H of width $\leq k$.*

Proof. The soundness and completeness of Algorithm 2 follow almost immediately from the soundness and completeness of Algorithm 1 together with the above explanations of the various optimisations. The only non-trivial part is that the last optimisation (i.e., the restriction of the search space for $\lambda(p)$) does not destroy the completeness of the algorithm. The remainder of the proof will concentrate on this aspect.

Assume that hypergraph H has an HD of width $\leq k$. Then the optimised `log-k-decomp` algorithm without the restriction on the search space for label λ_p (on line 23) returns the overall result *true*. This is due to the fact that, as was argued above, the other optimisations mentioned in this section do not affect the completeness of the algorithm. Now consider a recursive call of function `Decomp` and suppose that it returns true if the restriction on the search space for label λ_p is dropped. Of course, if the value *true* is returned in one of the base cases (lines 6 or 8) or if λ_c turns out to be the λ-label of the root node of the current HD-fragment (and *true* is returned on line 22), then the restriction of the search space for λ_p has no effect at all. Hence, the only interesting case to consider is that the "ParentLoop" (lines 23–47) is indeed executed.

Let λ_p be the λ-label chosen on line 23 if no restriction is imposed on the search space. We claim that we may remove from λ_p all edges that have an empty intersection width $\bigcup \lambda_c$ without altering the control flow of this particular execution of function Decomp. Actually, it suffices to show that we may remove *one* edge e with an empty intersection width $\bigcup \lambda_c$ from λ_p without altering the control flow of this particular execution of function Decomp. Then the claim follows by an easy induction argument.

So suppose that λ_p contains at least one edge e such that $e \cap \bigcup \lambda_c = \emptyset$, and let $\lambda'_p = \lambda_p \setminus \{e\}$. An inspection of the code of the "ParentLoop" reveals that it suffices to show that this elimination of edge e from λ_p leaves $comp_{down}$ unchanged. Indeed, if $comp_{down}$ is still a $[\lambda'_p]$-component, say, the ith $[\lambda'_p]$-component, then the if-condition on line 25 is true. Of course, there can be only one $[\lambda'_p]$-component satisfying the condition $comps_p[i]| > \frac{|H'.E|+|H'.Sp|}{2}$. Hence, on line 26, for this particular i, exactly the same value is assigned to $comp_{down}$ for λ'_p as for λ_p. But then also χ_c on line 29 gets the same value as without the restriction on the search space of λ_p. Consequently, also the $[\chi_c]$-components computed on line 35 and the parameters supplied to the recursive calls of function Decomp (on lines 39 and 45) remain the same as without the restriction on the search space. Hence, function Decomp will ultimately return the value *true* also if we choose λ'_p on line 23.

It remains to show that λ_p and λ'_p indeed give rise to the same component $comp_{down}$. To avoid confusion, let us write $comp_{down}$ to denote a $[\lambda_p]$-component and $comp'_{down}$ to denote a $[\lambda'_p]$-component. Let $comp_{down}$ be the unique $[\lambda_p]$-component that satisfies the condition $|comps_p[i]| > \frac{|H'.E|+|H'.Sp|}{2}$ on line 25. We have $\lambda'_p \subseteq \lambda_p$. Decreasing a set can only increase the corresponding components. Hence, there exists a $[\lambda'_p]$-component, call it $comp'_{down}$ with $comp_{down} \subseteq comp'_{down}$. We have to show that $comp_{down} = comp'_{down}$ holds.

The set $comp_{down}$ consists of the edges and special edges of the $[\chi_c]$-components contained in $comp_{down}$ (denoted as $comps_c$ in the algorithm), and the edges and special edges covered by χ_c. Let us refer to these $[\chi_c]$-components as $C_1, \ldots, C_\ell$. By Corollary 3.12, $comp_{down}$ is both a $[\lambda(p)]$-component and a $[\chi(p)]$-component. Hence, by Lemma 3.11, these $[\chi(c)]$-components are at the same time the $[\lambda_c]$components contained in $comp_{down}$. And the edges and special edges covered by χ_c are of course also covered by $\bigcup \lambda_c$. Likewise, $comp'_{down}$ consists of the (special) edges of the $[\lambda_c]$components contained in $comp'_{down}$ plus the (special) edges covered by λ_c.

By $comp_{down} \subseteq comp'_{down}$, all $[\lambda_c]$-components $C_1, \ldots, C_\ell$ contained in $comp_{down}$ are of course also contained in $comp'_{down}$. We have to show that there is no further $[\lambda_c]$-component contained in $comp'_{down}$. Assume to the contrary that there exists a $[\lambda_c]$-component C' in $comp'_{down}$ such that C' is not in $comp_{down}$. By definition, $comp_{down}$ is $[\lambda_p]$connected while $comp'_{down}$ is $[\lambda'_p]$connected. Hence, there exist (possibly special) edges $f' \in C'$ and $f \in C_i$ for some $i \in \{1, \ldots, \ell\}$, such that there is a path π (represented as a sequence of edges) with $\pi = (f_0, f_1, \ldots, f_m)$, such that $f = f_0$, $f' = f_m$, and $(f_\alpha \cap f_{\alpha+1}) \setminus \bigcup \lambda'_p \neq \emptyset$ for every $\alpha \in \{0, \ldots, m-1\}$. W.l.o.g., choose f, f', and π such that m is minimal. Since f and f' are not $[\lambda_p]$-connected, there exists α with $f_\alpha \cap f_{\alpha+1} \cap e \neq \emptyset$ while $(f_\alpha \cap f_{\alpha+1}) \setminus \bigcup \lambda_p = \emptyset$.

Since all (special) edges in $comp'_{down}$ are either in some $[\lambda_c]$-component contained in $comp'_{down}$ or covered by $\bigcup \lambda_c$, and since we are assuming that π is of minimal length, the path π starts with f in some $[\lambda_c]$-component C_i, possibly goes through $\bigcup \lambda_c$ and ends with f' in component C'. Recall that e was chosen such that $e \cap \bigcup \lambda_c = \emptyset$. Hence, the edges f_α and $f_{\alpha+1}$ cannot be covered by $\bigcup \lambda_c$. By our assumption that π has minimal length, we can also exclude the case that both f_α and $f_{\alpha+1}$ are in C'. Hence, at least one of f_α and $f_{\alpha+1}$ must be in C_i. In other words, $e \cap C_i \neq \emptyset$. Hence, also $e \cap comp_{down} \neq \emptyset$. However, by the check on line 32 in Algorithm 2, we know that $V(comp_{down}) \cap \bigcup \lambda_p \subseteq \bigcup \lambda_c$. This contradicts the assumption that $e \in \lambda_p$ and $e \cap \bigcup \lambda_c = \emptyset$. □

8 IMPLEMENTATION AND EVALUATION

We report now on the empirical results obtained for our implementation of the log-k-decomp algorithm. Our experiments are based on the HyperBench benchmark from [12], which was already used for the evaluation of previous decomposition algorithms, notably NewDetKDecomp [12] (an enhanced re-implementation of det-k-decomp [27]) and HtdLEO [31].

The goal of the experiments is to determine the exact hypertree width of as many instances as possible. We compare here the performance of three different decomposition methods, namely NewDetKDecomp [12], HtdLEO [31], and our implementation of log-k-decomp. Note that while the tested implementations include the capability to compute GHDs or FHDs, we only consider the computation of HDs in our experiments here. Our new implementation of log-k-decomp is based on the open-source code of BalancedGo [26], a parallel algorithm for computing GHDs.

Parallel Implementation. For our experiments, we implemented log-k-decomp including all of the optimisations presented in Section 7. As discussed above, a crucial aspect of our algorithm design is that the use of balanced separators allows us to recursively split the problem into smaller subproblems. The subproblems are independent of each other and are therefore processed in parallel by our implementation. Furthermore, following observations made in Reference [26], our implementation also executes the search for balanced separators in parallel by partitioning the search space effectively.

The full raw data of our experiments is publicly available [18], as is the source code of our implementation[1] of log-k-decomp.

8.1 Benchmark Instances and Setting

For the evaluation, we use the benchmark library HyperBench [12]. It contains 3,648 hypergraphs underlying CQs and CSPs from various sources in industry and the literature and is commonly used to evaluate decomposition algorithms.

A number of our experiments are performed over the full set of HyperBench instances. However, for some of our experiments it is more meaningful to restrict them to exclude hypergraphs that are, roughly speaking, too small or have high width. Small instances benefit only marginally from algorithmic improvements or parallelism, while very high width is of less algorithmic interest as it exponentially effects algorithms that make use of decompositions. Hence, we propose to exclude such instances in some cases to make more relevant observations. For the purpose of creating this restricted set of instances, we focus on instances with more than 50 edges and vertices that are known to have hypertree width at most 6. There are 465 instances in HyperBench that satisfy these conditions; we will refer to them throughout this section as HB_{large}.

The instances are available at http://hyperbench.dbai.tuwien.ac.at and also in the published raw data of our experiments [18].

Hardware and Software. Our implementation is written in the programming language Go using version 1.14, and we will refer to it as log-k-decomp. We will give more details below on how it is configured for the experiments we report in Section 8.4. The hardware we use for the evaluation is a cluster of 12 nodes, using Ubuntu 16.04.1 LTS, with Linux kernel 4.4.0-184-generic, GCC version 5.4.0. Each node has a 12 core Intel Xeon CPU E5-2650 v4, clocked at 2.20 GHz and using 264 GB of RAM.

Setup of Experiments. For the current extended version of this work, we used the following time and memory limits: a timeout of one hour is used uniformly for all systems and we limited

[1] https://github.com/cem-okulmus/log-k-decomp

available RAM to 1 GB for NewDetKDecomp, to 24 GB of RAM for HtdLEO, and to 40 GB of RAM for log-k-decomp when using the hybrid version (to be discussed in detail in Section 8.3). The differences are explained by choosing for each system a limit such that no "out of memory" error was reported on our machines and it thus roughly reflects an upper bound on the needed amount of memory for each system to run every single instance in our used benchmark. For log-k-decomp, each run needs two inputs: a hypergraph H and the width parameter $k \geq 1$. For these tests, we use width parameters in the range $[1, 10]$. A peculiarity of HtdLEO is that it needs no width parameter, since it directly tries to find an optimal solution. We use the HTCondor system [32] to execute the tests. HTCondor is also used to manage the limits to memory and number of cores accessible by the test instances.

Throughout this section, we will be interested in two key metrics. First, the number of *solved* instances, by which we mean instances for which an optimal (i.e., minimal width) hypertree decomposition is found and proven optimal. Second, the computation time that is necessary to compute the optimal width decomposition, which we will refer to as the *runtime*. The "optimal (hypertree-)width" is always understood as the *known* optimal width, using all sources of information available to us: the HyperBench website, past published experiments, and our own experiments.

Importantly, this means that average runtimes are taken only over the instances that the respective algorithm is able to solve, while timed out instances are not considered in the runtime calculation.

8.2 Implementation Improvements

Parts of this article were published earlier in Reference [17] where we reported on preliminary versions of some of the experiments reported here. We have since improved the implementation of the algorithm in various critical aspects and in consequence we wish to note that the results in this article show significant improvement over those reported in Reference [17]. Here we briefly summarise these changes.

Detailed low-level analysis of the previous implementation revealed that a significant portion of the time was spent on memory management in the Go running time. Go uses a garbage collector to automatically manage dynamically allocated memory and both frequent allocation of memory and garbage collection itself made up a substantial percentage of the overall execution time. Critically, the proportion of time spent in memory management increased as the number of parallel execution threads increased. We observed that the number of memory allocations scaled linearly with the number of parallel threads, thus also triggering the garbage collection more frequently in proportion to the number of threads. Our analysis showed that the ever-increasing time spent with memory management was a key factor of the diminishing returns that we observed when scaling our implementation to more than four CPUs in the preceding conference paper [17].

To alleviate the high costs of memory management, we improve our implementation in two main aspects. First, through various low-level optimisations we greatly reduce the number of memory allocations performed when computing the $[S]$-components for a separator S. This is the key subroutine in the check for balanced separation (cf. Algorithm 7.1) and our changes reduce the number of overall memory allocations by orders of magnitude. In addition, we configure the garbage collector trigger to adapt to the proportion to the number of CPUs used. That is, when using the n CPU cores, we set the garbage collector trigger condition to be n times less sensitive than the standard setting, thus approximately achieving the same overall frequency of garbage collection for any number of CPUs.

These two measures together, significantly reducing memory allocations overall as well as throttling the garbage collector so that it does not dominate the execution time as the number of used CPUs increases, help us achieve both better performance overall—we manage to solve more than

150 new instances optimally within the timeout of one hour and crucially also improve scaling behaviour when using more CPUs.

8.3 Hybrid Approaches

While our algorithm has desirable properties for parallelisation (in particular, the logarithmically bounded recursion depth established in Theorem 4.1), this comes at the cost of some overhead when compared to simpler methods, in particular det-k-decomp. Especially on small and simple instances, the restriction to balanced separators may act as a detriment to performance that outweighs its benefits for parallelisation and its effect on severely restricting the search space.

To balance these considerations overall in practice, we therefore also consider hybrid variants of our implementation. Intuitively, we want to use log-k-decomp as long as the subproblems are still complex, but once they become simple, we want to switch to an algorithm that is better suited for those cases. For the simpler algorithm, det-k-decomp is the natural choice, as it performs very well on small instances as is shown in Reference [12], where an implementation of det-k-decomp is provided as part of NewDetKDecomp. To determine when the switch is made, we implement two simple metrics to capture the complexity of a hypergraph:

EdgeCount In EdgeCount we simply use the number of edges of the hypergraph $|E(H)|$ as the measure of complexity.

WeightedCount The WeightedCount metric is characterised by the formula $|E(H)|\frac{k}{avg_{e\in E(H)}|e|}$ where k is the width parameter of the algorithm. The additional factor compared to EdgeCount is best understood as two separate additional weightings. Higher width implies more complex structure and hence we expect more complexity per edge. However, if edges are on average larger, then it becomes easier to find covers and we therefore also inversely weight by the average cardinality of the edges.

We investigate the effectiveness of these metrics through a series of experiments. In particular, we conduct experiments with both metrics and different thresholds for when to switch from log-k-decomp to det-k-decomp. To be precise, for a metric m and threshold T, log-k-decomp is executed for a subproblem with hypergraph H_i as long as $m(H_i) \geq T$. If $m(H_i) < T$, then we switch to an implementation of det-k-decomp written from scratch as part of the code base of log-k-decomp. A similar strategy is also proposed in Reference [26] in the context of BalancedGo. However, in that system no metric for the complexity of a subproblem is employed, but rather the switch to det-k-decomp is always performed at a fixed recursion depth.

Our results for the experiments on HB_{large} are summarised in Table 1. Methods WeightedCount and EdgeCount refer to log-k-decomp with the respective metric used for hybridisation. The threshold column refers to parameter T in the discussion above. The experiments for all log-k-decomp hybrid methods had access to 12 cores, the experiments for NewDetKDecomp and HtdLEO used only 1 core each, since they do not support parallelism.

Overall, WeightedCount clearly performs best, especially in the number of solved instances. For thresholds 400 and 600, approximately 90% of the 465 large instances from HB_{large} are solved. This constitutes a significant improvement over the 37% and 60% achieved by NewDetKDecomp and HtdLEO, respectively. Note that despite solving more instances—for which det-k-decomp and HtdLEO timed out—the runtime is also at least 3 times lower for WeightedCount. This is surprising, as we do not consider timed-out instances in our average runtime calculations.

One surprising observation from the table is that the differences in performance between different thresholds are much smaller for WeightedCount than for EdgeCount. Further investigation suggests that this is due to the WeightedCount metric decreasing much more rapidly as hypergraphs become simpler. At the same time, for subproblems that fall in the range between 200 and

Table 1. Study of Two Hybrid Methods of log-k-decomp on HB_{large} and a Comparison with NewDetKDecomp and HtdLEO for Reference

Method	Threshold	Solved	Avg. runtime (sec.)
WeightedCount	200	395	92.15
WeightedCount	400	**411**	93.53
WeightedCount	600	410	**87.86**
EdgeCount	20	171	130.0
EdgeCount	40	219	145.09
EdgeCount	80	292	117.33
NewDetKDecomp [12]	—	174	318.93
HtdLEO [31]	—	277	779.39

Bolded entries represent the best value among the compared methods, where 'best' is understood as the highest value in the solved column and the lowest value in the run time column.

600, the performance of switching to det-k-decomp immediately is roughly the same (on average) as the performance of continuing with log-k-decomp for one or two more steps.

While EdgeCount performs worse than WeightedCount, we can still see a clear improvement over the state-of-the-art methods NewDetKDecomp and HtdLEO. Especially the significant improvement over det-k-decomp is important to observe as it clearly demonstrates the benefits of our hybrid approach. Recall that when we split our problem into balanced subproblems, each subproblem is then solved independently in parallel. In the hybrid variant, we will thus eventually execute our implementation of the det-k-decomp algorithm on multiple subproblems in parallel, i.e., we can use an inherently single-threaded algorithm effectively in parallel, because we are able to create balanced subproblems.

8.4 Empirical Evaluation

We compare the aforementioned hybrid version of the log-k-decomp algorithm with the two state-of-the-art implementations for finding HDs: NewDetKDecomp [12] and HtdLEO [31].

Our results are summarised in Table 2, distinguishing the hypergraphs in the HyperBench benchmark by size and origin.[2] We distinguish between two main categories, hypergraphs that are derived from applications and hypergraphs that were synthetically generated. In each group, we report our results split by the number of edges $|E|$ in the instance. Note that the group $|E| > 100$ of instances with more than 100 edges is empty for the Application case and thus omitted from the table. *Instances in Group* reports the number of instances in each such group. For each algorithm and each group of instances; we list the number of solved instances (*#solved*) and statistics over the runtimes (*avg*, *max*, *stdev*). Times are all in seconds and rounded to a single digit after the comma. Results over all groups are given in the last row titled "Total."

As mentioned above, some care is required when comparing times between algorithms. While NewDetKDecomp has low average time overall, this is partly due to solving fewer instances. The data therefore demonstrate that, in general, NewDetKDecomp either solves an instance quickly or fails to find an optimal width decomposition before timing out. Overall, we see that despite solving significantly more instances than its competitors, runtimes for log-k-decomp overall are comparable with NewDetKDecomp and noticeably lower than for HtdLEO.

[2]HyperBench instances are often categorised more fine-grained in terms of their origin (cf. Reference [12]). For our experiments, we have found the direct effect of hypergraph size to be more informative and therefore report our results in this way instead.

Table 2. Comparison of Prior Methods and log-k-decomp: Number of Cases Solved and Runtimes (Seconds) to Find Optimal-width HDs

			Hypertree Decomposition Methods							
Origin of Instances	Size of Instances	Instances in Group	NewDetKDecomp [12]				HtdLEO [31]			
			#solved	avg	max	stdev	#solved	avg	max	stdev
Application	$75 < \|E\| \leq 100$	405	97	31.8	3298.0	248.4	65	809.5	3156.6	735.2
	$50 < \|E\| \leq 75$	514	276	10.6	1906.0	104.7	448	250.0	3281.5	409.3
	$10 < \|E\| \leq 50$	369	253	0.0	0.0	0.0	237	60.1	1017.9	150.3
	$\|E\| \leq 10$	915	906	0.0	0.0	0.0	876	56.6	1427.1	155.0
Synthetic	$\|E\| > 100$	66	18	0.3	7.0	1.0	13	734.0	2507.1	711.7
	$75 < \|E\| \leq 100$	422	87	71.7	3467.0	379.4	312	1045.2	3591.1	1287.0
	$50 < \|E\| \leq 75$	215	38	18.8	1593.0	141.9	212	101.7	2560.1	246.1
	$10 < \|E\| \leq 50$	647	290	58.6	3240.0	337.2	303	412.2	3597.4	850.2
	$\|E\| \leq 10$	95	**95**	0.0	0.0	0.0	78	28.8	218.5	41.5
Total	—	3648	2060	23.1	3467.0	208.0	2544	280.2	3597.4	676.7

			Hypertree Decomposition Methods			
Origin of Instances	Size of Instances	Instances in Group	log-k-decomp Hybrid			
			#solved	avg	max	stdev
Application	$75 < \|E\| \leq 100$	405	**317**	83.8	3325.9	364.6
	$50 < \|E\| \leq 75$	514	**470**	3.3	1506.5	69.5
	$10 < \|E\| \leq 50$	369	**253**	0.0	0.1	0.0
	$\|E\| \leq 10$	915	**915**	0.0	0.0	0.0
Synthetic	$\|E\| > 100$	66	**35**	14.7	295.1	54.7
	$75 < \|E\| \leq 100$	422	**313**	73.9	3565.6	395.0
	$50 < \|E\| \leq 75$	215	**215**	1.1	103.2	7.2
	$10 < \|E\| \leq 50$	647	**637**	21.6	3402.3	179.1
	$\|E\| \leq 10$	95	**95**	0.0	0.0	0.0
Total	—	3648	**3250**	20.6	3565.6	190.9

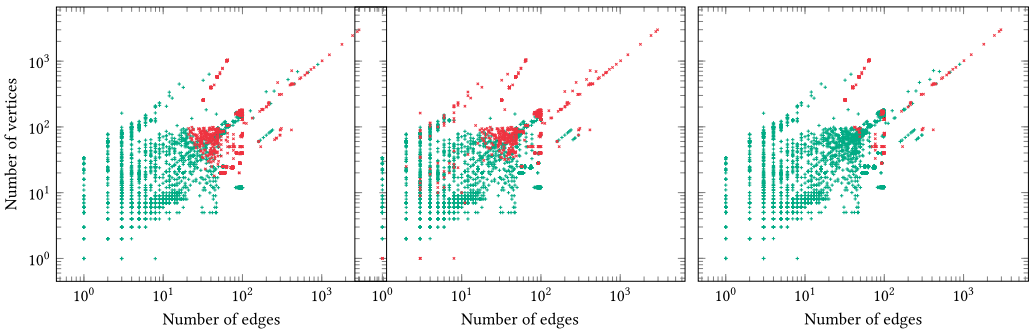
Bolded entries in the solved column represent the highest value achieved among the three compared systems for a given group, thus marking the best achieved result.

HtdLEO *with 10 Hour Timeout.* The method that is used in HtdLEO fundamentally differs from the search algorithm here. In HtdLEO, the problem is encoded to the **SAT modulo theories (SMT)** setting and the final solving step is handed off to standard SMT solvers. The encoding in HtdLEO is constructed in such a way that no width parameter is handed to the solver; rather, the encoding will always try to return the optimal width as its solution. This is a significant difference to the parameterised search implemented in log-k-decomp (but also previous algorithms such as det-k-decomp and BalancedGo). This difference makes it naturally difficult to compare runtimes and the number of optimal solutions directly. To provide a fuller picture we add here Table 3. In the table, we present the results from running the same experiments with HtdLEO but with the timeout increased to 10 hours. This increase in timeouts naturally makes the average runtimes difficult to compare to the values in Table 2. However, importantly, we see that the increase in solved instances is also moderate, and overall log-k-decomp still solves significantly more instances than HtdLEO with 10 hours of maximum runtime.

It may be of further interest how these numbers compare to the performance of state-of-the-art algorithms for finding generalised hypertree decompositions. The results reported for BalancedGo [26] (on a comparable system, according to the description of the experiments) show that the best method there solves only 2,924 instances optimally for *ghw*. In contrast, when considering the same set of instances, log-k-decomp manages to solve 3,250 of the instances

Table 3. Extension of Table 2 with Runtimes for HtdLEO Extended to 10 Hours

Origin of Instances	Size of Instances	Instances in Group	HtdLEO [31] *10 Hour Run* #solved	Changes 1 hour run
Application	$75 < \|E\| \leq 100$	405	94	+ 29
	$50 < \|E\| \leq 75$	514	461	+ 13
	$10 < \|E\| \leq 50$	369	237	± 0
	$\|E\| \leq 10$	915	876	± 0
Synthetic	$\|E\| > 100$	66	13	± 0
	$75 < \|E\| \leq 100$	422	360	+ 48
	$50 < \|E\| \leq 75$	215	214	+ 2
	$10 < \|E\| \leq 50$	647	433	+130
	$\|E\| \leq 10$	95	78	± 0
Total	—	3648	2766	+222



(a) NewDetKDecomp (b) HtdLEO (c) log-k-decomp (Hybrid)

Fig. 6. Comparison of solved instances (green) and unsolved instances (red), relative to their edge and vertex size.

tested there optimally. When looking at the average running times over all instances, we get 25.76 s for BalancedGo, but 20.6 s for log-k-decomp. Thus, it solves considerably more instances and does so faster on average. Furthermore, in none of the cases where BalancedGo finds the optimal ghw is it lower than the optimal hw. In other words, in practice, the additional complexity of GHDs compared with HDs is not compensated by achieving lower width (even if, in theory, no better upper bound on the hw than $hw \leq 3 \cdot ghw + 1$ is known [2]).

In our experiments, we also observe that for low widths—i.e., cases where using HDs is most promising in practice—log-k-decomp is essentially able to solve all instances. In particular, of the 2,947 instances with width at most 6, log-k-decomp solves 2,941 (99.8%) instances. In contrast, NewDetKDecomp and HtdLEO time out on 488 and 919, respectively, of those instances. This suggests that log-k-decomp can be a solid foundation for the integration of HDs in practice going forward.

To gain further insight into the performance of each algorithm with respect to solving instances optimally, we investigate the size of solved and unsolved instances. To this end, Figure 6 provides (logarithmic) scatter plots for each of the three algorithms in our tests. In each plot, each instance is positioned according to its number of vertices and edges. Solved instances for each algorithm are drawn in green while unsolved instances are drawn in red.

The plots show that our intuition holds true in that solving large instances (in both axis) significantly benefits from using log-k-decomp. Most of the remaining hypergraphs are either extremely large, containing thousands of edges and vertices, or belong to very specific CSP classes that we

Table 4. Comparison of the Decomposition Methods by How Many Instances Were Solved for a Specific Width

Width	Virtual Best	NewDetKDecomp	HtdLEO	log-k-decomp (Hybrid)
1	709	677	649	**709**
2	595	586	567	**595**
3	310	**310**	273	**310**
4	386	379	321	**386**
5	450	38	341	**450**
6	496	28	307	**491**
7	249	9	16	**233**
8	116	1	**69**	47
9	29	0	1	**28**
10	1	0	0	**1**

Bolded entries represent the highest value achieved among the three compared systems for a given width, thus marking the best achieved result.

know to have very high width (significantly beyond the width 10 limit used in our experiments) through graph-theoretic arguments.

We want to analyse for how many instances of HyperBench each decomposition method could determine its width and specifically focus on how well it fares as the width increases. Note that HtdLEO is unique in this respect, since it determines the optimal width right away. Thus, if we ask for how many instances HtdLEO could determine if its width is ≤5, for example, then we are really just counting how many timeouts there were in general. For the parametrised decomposition methods, however, this question does give us new insights into how its run timescales when looking for decompositions of larger or smaller width.

For this purpose, we first need the concept of the "Virtual Best" method. This notion simply aggregates the results of all considered methods and shows for how many instances of HyperBench we know their hw. We can see in Table 4 how each of the three methods fares when compared against this virtual best method. For widths up to 7, the Hybrid log-k-decomp is unbeaten, solving all known instances, as well as solving many of them exclusively.

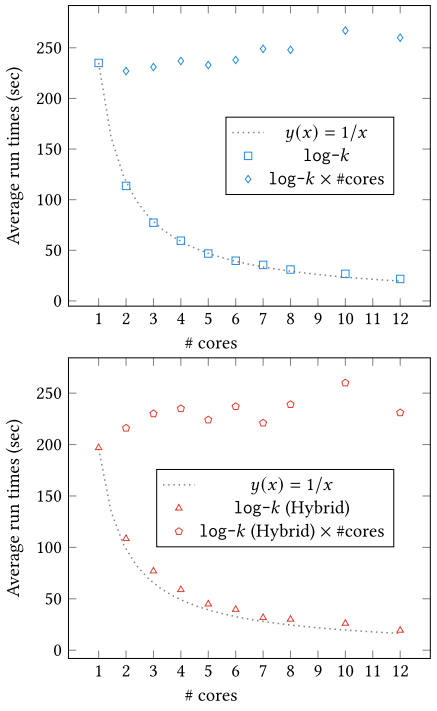
To provide a more detailed analysis, we also compare for how many instances of hw up to 6, each method can determine whether an instance has hw of lower than the given number by finding an HD of such a width or determining that no such HD can exist. Note that this does not require proving optimality. We can see the results in Table 5. We can see that both log-k-decomp and log-k-decomp (Hybrid) are very good at this, with the Hybrid determining for 3,437 (over 94% of) instances whether they have $hw \leq 6$. If we limit ourselves to $hw \leq 5$, then it determines the question for 3,612 or 99% of instances.

We perform another set of experiments over the instances in HB_{large} to verify our claims that log-k-decomp is well suited for parallelisation. For $1 \leq n \leq 12$, we observe the time taken to find and verify the optimal width of an instance using n CPU cores. We report on the times to find these optimum widths averaged over all instances in HB_{large} in Figure 7. We will briefly note that the *ideal* scaling function one could expect here is $\frac{1}{x}$, where we normalise the runtime at the sequential (1 core) case as 1, and divide by the number of cores x. To make the comparison against this ideal function easier, we added it to Figure 7 as a dotted line. Another way to illustrate this ideal scaling function is by taking the time when using x cores, and multiplying it with x. This should produce a constant function in the ideal case. We have also added the runtimes multiplied by number of cores to Figure 7. We observe near optimal speedups (in the sense described above)

Table 5. Comparison of the Decomposition Methods by the Upper Bounds It Could Provide

Problem to solve	Virtual Best	log-k-decomp (Hybrid)	NewDetKDecomp	log-k-decomp
$hw \leq 1$	3,648	**3,648**	3,616[3]	**3,648**
$hw \leq 2$	3,648	**3,648**	3,631	**3,648**
$hw \leq 3$	3,641	**3,641**	3,355	3,567
$hw \leq 4$	3,626	**3,626**	2,391	3,178
$hw \leq 5$	3,617	**3,612**	2,485	2,924
$hw \leq 6$	3,457	**3,437**	2,897	2,349

Note that HtdLEO is not being explicitly considered here, since it directly computes the optimal width. Thus it would have the number 2,544—its number of solved instances—in each row.
Bolded entries represent the highest value achieved among the three compared systems for a given problem, thus marking the best achieved result.



#cores	log-k	× #cores	log-k (hybrid)	× #cores
1	235.06	235.06	196.89	196.89
2	113.61	227.22	108.44	216.88
3	77.27	231.81	76.95	230.85
4	59.48	237.92	58.81	235.24
5	46.80	234.00	44.81	224.05
6	39.69	238.14	39.60	237.60
7	35.59	249.13	31.69	221.83
8	31.00	248.00	29.96	239.68
10	26.75	267.50	26.08	260.80
12	21.67	260.04	19.31	231.72

Method	Timeouts
log-k (Hybrid) (1 core)	94
log-k (1 core)	549

Method	solved > 100 ms	solved ≤ 100 ms
log-k (Hybrid) (1 core)	1137	1094
log-k (1 core)	1195	581

Fig. 7. Study of log-k-decomp scaling behaviour w.r.t. the number of processing cores used. For computing the averages, we only consider instances (combination of graph and width) where the sequential case (1 core) took more than 100 ms to solve. This study involved $465 * 5 = 2,325$ instances, namely every combination of an instances from HB_{large} and a width between 2 and 6.

up to 8 cores, from about 235 s on 1 core to 31 s for 4 cores for log-k-decomp. This behaviour is expected, since our parallelisation strategy relies on dividing up the search space for bounded separators uniformly over the the available cores. Since this requires no communication between

[3]We note that the reason NewDetKDecomp fails to determine acyclicity (i.e., whether $hw \leq 1$) for all graphs is due to a bug where it will output a HD of width 2 instead of one of width 1 when given certain acyclic graphs. While of low practical interest, as determining acyclicity is a trivial problem, it is still the case that NewDetKDecomp in its current form fails to determine all acyclic graphs of HyperBench correctly.

threads or other overhead that depends on the degree of parallelisation, the key task of searching for balanced separators scales evenly in the number of cores. In instances where the search for separators dominates the runtime, such as negative instances where the full search space is explored, analysis of our algorithm therefore predicts effectively ideal scaling of performance. In the data from Figure 7, we observe diminishing returns in the rate of improvement of average runtime starting from 8 cores, though we still see that the average runtime is reduced. Very similar scaling can be observed for our Hybrid version.

9 CONCLUSION AND OUTLOOK

In this article, we have introduced a novel algorithm, `log-k-decomp`, for computing hypertree decompositions. Based on new theoretical insights and results on HDs, we were able to propose an algorithm that constructs decompositions in arbitrary order (rather than, e.g., in a strict top-down manner) while achieving a *balanced* separation into subproblems. In this way, we have obtained a logarithmic bound on the recursion depth of our algorithm, making it particularly well suited for parallelisation. We evaluated an implementation of `log-k-decomp` through experimental comparison with the state of the art. On the standard benchmark for hypertree decomposition, we are able to achieve clear improvements both in the number of solved instances and in the time required to solve them.

In combination, our theoretical results and experiments demonstrate that `log-k-decomp` achieves our goal of effective parallel HD computation. We believe that the performance improvements, especially on hypergraphs with low width and for large hypergraphs, lay a strong foundation for more widespread adoption of hypertree decompositions in practice, e.g., for complex query execution in high-performance database applications.

With HD computation for large and complex hypergraphs becoming practically feasible, one of the key challenges that block the use of HDs is quickly becoming less problematic. We therefore consider full integration of hypertree decompositions into existing database systems and constraint solvers to be a natural next step in this line of research.

Experiments suggest that there is significant potential in the study of metrics for hybrid approaches. In particular, how can we decide effectively when to switch from the balanced separation of `log-k-decomp` to the greedy heuristic-guided method underlying `det-k-decomp`. This motivates a more in-depth study of hybridisation metrics in the future.

ACKNOWLEDGMENTS

We are very grateful to the anonymous reviewers for the careful reading and their valuable suggestions that helped to greatly improve the article.

REFERENCES

[1] Christopher R. Aberger, Andrew Lamb, Susan Tu, Andres Nötzli, Kunle Olukotun, and Christopher Ré. 2017. EmptyHeaded: A Relational Engine for Graph Processing. *ACM Trans. Database Syst.* 42, 4 (2017), 20:1–20:44. https://doi.org/10.1145/3129246

[2] Isolde Adler, Georg Gottlob, and Martin Grohe. 2007. Hypertree width and related hypergraph invariants. *Eur. J. Comb.* 28, 8 (2007), 2167–2181.

[3] Foto N. Afrati, Manas R. Joglekar, Christopher Ré, Semih Salihoglu, and Jeffrey D. Ullman. 2017. GYM: A multiround distributed join algorithm. In *Proceedings of the 20th International Conference on Database Theory (ICDT 2017), LIPIcs, Vol. 68*. Schloss Dagstuhl, 4:1–4:18. https://doi.org/10.4230/LIPIcs.ICDT.2017.4

[4] Dmitri Akatov. 2010. *Exploiting Parallelism in Decomposition Methods for Constraint Satisfaction*. Ph. D. Dissertation. University of Oxford, UK.

[5] Hans L. Bodlaender. 1989. Improved self-reduction algorithms for graphs with bounded treewidth. In *Proceedings of the International Workshop on Graph-Theoretic Concepts in Computer Science (WG'89), Lecture Notes in Computer Science, Vol. 411*, Manfred Nagl (Ed.). Springer, 232–244. https://doi.org/10.1007/3-540-52292-1_17

[6] Hans L. Bodlaender and Torben Hagerup. 1998. Parallel algorithms with optimal speedup for bounded treewidth. *SIAM J. Comput.* 27, 6 (1998), 1725–1746. https://doi.org/10.1137/S0097539795289859

[7] Ashok K. Chandra and Larry J. Stockmeyer. 1976. Alternation. In *Proceedings of the 17th Annual Symposium on Foundations of Computer Science*. 98–108.

[8] Stephen A. Cook. 1985. A taxonomy of problems with fast parallel algorithms. *Inf. Contr.* 64, 1-3 (1985), 2–22. https://doi.org/10.1109/SFCS.1976.4

[9] M. Ayaz Dzulfikar, Johannes Klaus Fichte, and Markus Hecher. 2019. The PACE 2019 Parameterized Algorithms and Computational Experiments Challenge: The Fourth Iteration (Invited Paper). In *Proceedings of the 14th International Symposium on Parameterized and Exact Computation (IPEC'19)*. 25:1–25:23.

[10] Ronald Fagin. 1983. Degrees of acyclicity for hypergraphs and relational database schemes. *J. ACM* 30, 3 (1983), 514–550. https://doi.org/10.1145/2402.322390

[11] Johannes Klaus Fichte, Markus Hecher, Neha Lodha, and Stefan Szeider. 2018. An SMT Approach to Fractional Hypertree Width. In *Proceedings of the 24th International Conference on Principles and Practice of Constraint Programming (CP'18)*. 109–127.

[12] Wolfgang Fischl, Georg Gottlob, Davide Mario Longo, and Reinhard Pichler. 2021. HyperBench: A benchmark and tool for hypergraphs and empirical findings. *ACM J. Exp. Algor.* 26 (2021), 1.6:1–1.6:40. https://doi.org/10.1145/3440015

[13] Lucantonio Ghionna, Luigi Granata, Gianluigi Greco, and Francesco Scarcello. 2007. Hypertree decompositions for query optimization. In *Proceedings of the 23rd International Conference on Data Engineering (ICDE'07)*. IEEE Computer Society, 36–45.

[14] Lucantonio Ghionna, Gianluigi Greco, and Francesco Scarcello. 2011. H-DB: A hybrid quantitative-structural SQL optimizer. In *Proceedings of the 20th ACM Conference on Information and Knowledge Management (CIKM 2011)*. ACM, 2573–2576. https://doi.org/10.1145/2063576.2064023

[15] Georg Gottlob and Gianluigi Greco. 2013. Decomposing combinatorial auctions and set packing problems. *J. ACM* 60, 4 (2013), 24:1–24:39. https://doi.org/10.1145/2505987

[16] Georg Gottlob, Gianluigi Greco, and Francesco Scarcello. 2005. Pure nash equilibria: Hard and easy games. *J. Artif. Intell. Res.* 24 (2005), 357–406. https://doi.org/10.1613/jair.1683

[17] Georg Gottlob, Matthias Lanzinger, Cem Okulmus, and Reinhard Pichler. 2022. Fast parallel hypertree decompositions in logarithmic recursion depth. In *Proceedings of the 41st ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems (PODS'22)*. ACM, 325–336. https://doi.org/10.1145/3517804.3524153

[18] Georg Gottlob, Matthias Lanzinger, Cem Okulmus, and Reinhard Pichler. 2023. Experimental data for log-k-decomp. Zenodo. https://doi.org/10.5281/zenodo.7180787

[19] Georg Gottlob, Matthias Lanzinger, Reinhard Pichler, and Igor Razgon. 2020. Fractional covers of hypergraphs with bounded multi-intersection. In *Proceedings of the 45th International Symposium on Mathematical Foundations of Computer Science (MFCS'20), LIPIcs, Vol. 170*. Schloss Dagstuhl, 41:1–41:14.

[20] Georg Gottlob, Matthias Lanzinger, Reinhard Pichler, and Igor Razgon. 2021. Complexity analysis of generalized and fractional hypertree decompositions. *J. ACM* 68, 5 (2021), 38:1–38:50. https://doi.org/10.1145/3457374

[21] Georg Gottlob, Nicola Leone, and Francesco Scarcello. 2000. A comparison of structural CSP decomposition methods. *Artif. Intell.* 124, 2 (2000), 243–282.

[22] Georg Gottlob, Nicola Leone, and Francesco Scarcello. 2001. The complexity of acyclic conjunctive queries. *J. ACM* 48, 3 (2001), 431–498.

[23] Georg Gottlob, Nicola Leone, and Francesco Scarcello. 2002. Computing LOGCFL certificates. *Theor. Comput. Sci.* 270, 1-2 (2002), 761–777.

[24] Georg Gottlob, Nicola Leone, and Francesco Scarcello. 2002. Hypertree decompositions and tractable queries. *J. Comput. Syst. Sci.* 64, 3 (2002), 579–627. https://doi.org/10.1006/jcss.2001.1809

[25] Georg Gottlob, Zoltán Miklós, and Thomas Schwentick. 2009. Generalized hypertree decompositions: NP-hardness and tractable variants. *J. ACM* 56, 6 (2009), 30:1–30:32. https://doi.org/10.1145/1568318.1568320

[26] Georg Gottlob, Cem Okulmus, and Reinhard Pichler. 2022. Fast and parallel decomposition of constraint satisfaction problems. *Constr. Int. J.* 27, 3 (2022), 284–326. https://doi.org/10.1007/s10601-022-09332-1

[27] Georg Gottlob and Marko Samer. 2008. A backtracking-based algorithm for hypertree decomposition. *ACM J. Exp. Algor.* 13 (2008). https://doi.org/10.1145/1412228.1412229

[28] Jens Lagergren. 1990. Efficient parallel algorithms for tree-decomposition and related problems. In *Proceedings of the 31st Annual Symposium on Foundations of Computer Science*. IEEE Computer Society, 173–182. https://doi.org/10.1109/FSCS.1990.89536

[29] Reinhard Pichler and Sebastian Skritek. 2013. Tractable counting of the answers to conjunctive queries. *J. Comput. Syst. Sci.* 79, 6 (2013), 984–1001. https://doi.org/10.1016/j.jcss.2013.01.012

[30] Walter L. Ruzzo. 1980. Tree-size bounded alternation. *J. Comput. Syst. Sci.* 21, 2 (1980), 218–235.

[31] André Schidler and Stefan Szeider. 2021. Computing optimal hypertree decompositions with SAT. In *Proceedings of the 30th International Joint Conference on Artificial Intelligence (IJCAI'21)*. 1418–1424. https://doi.org/10.24963/ijcai.2021/196

[32] Douglas Thain, Todd Tannenbaum, and Miron Livny. 2005. Distributed computing in practice: The Condor experience. *Concurr. Pract. Exp.* 17, 2-4 (2005), 323–356.

[33] Mihalis Yannakakis. 1981. Algorithms for acyclic database schemes. In *Proceedings of the 7th International Conference on Very Large Databases (VLDB'81)*. VLDB, 82–94.

Received 14 October 2022; revised 22 June 2023; accepted 4 October 2023