



Fast and parallel decomposition of constraint satisfaction problems

Georg Gottlob¹ · Cem Okulmus² · Reinhard Pichler²

Accepted: 23 April 2022 / Published online: 3 June 2022
© The Author(s) 2022

Abstract

Constraint Satisfaction Problems (CSP) are notoriously hard. Consequently, powerful decomposition methods have been developed to overcome this complexity. However, this poses the challenge of actually computing such a decomposition for a given CSP instance, and previous algorithms have shown their limitations in doing so. In this paper, we present a number of key algorithmic improvements and parallelisation techniques to compute so-called Generalized Hypertree Decompositions (GHDs) faster. We thus advance the ability to compute optimal (i.e., minimal-width) GHDs for a significantly wider range of CSP instances on modern machines. This lays the foundation for more systems and applications in evaluating CSPs and related problems (such as Conjunctive Query answering) based on their structural properties.

Keywords Constraint satisfaction · Hypergraphs · Structural decomposition methods · Parallel computing

1 Introduction

Many real-life tasks can be effectively modelled as CSPs, giving them a vital importance in many areas of Computer Science. As solving CSPs is a classical NP-complete problem, there is a large body of research to find tractable fragments. One such line of research focuses on the underlying *hypergraph structure* of a CSP instance. A key result in this area

This work was supported by the Austrian Science Fund (FWF):P30930-N35. 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).

✉ Cem Okulmus
cokulmus@dbai.tuwien.ac.at

Georg Gottlob
georg.gottlob@cs.ox.ac.uk

Reinhard Pichler
pichler@dbai.tuwien.ac.at

¹ University of Oxford, Oxford, UK

² TU Wien, Vienna, Austria

is that CSP instances whose underlying hypergraph is acyclic, can be solved in polynomial time [41]. Several generalisations of acyclicity have been identified by defining various forms of hypergraph *decompositions*, each associated with a specific notion of *width* [8, 18]. Intuitively, the width measures how far away a hypergraph is from being acyclic, with a width of 1 describing the acyclic hypergraphs.

In this work, we focus on Generalized Hypertree Decompositions (GHD) [20], and generalized hypertree width (*ghw*). Formally, we look at the following problem:

CHECKGHD

Input hypergraph $H = (V, E)$;

Parameter k ;

Output GHD of H of width $\leq k$ if it exists and answer ‘no’ otherwise.

The computation of GHDs is itself intractable in the general case, already for width = 2 [15]. However, for (hypergraphs of) CSPs with realistic restrictions, this problem becomes tractable for a fixed parameter k . One such restriction is the bounded intersection property (BIP), which requires that any two constraints in a CSP only share a bounded number of variables [15]. Indeed, by examining a large number of CSPs from various benchmarks and real-life applications, it has been verified that this intersection of variables tends to be small in practice [14]. In that work, over 3,000 instances of hypergraphs of CSPs and also of Conjunctive Queries (CQs) were examined and made publicly available in the HyperBench benchmark at <http://hyperbench.dbai.tuwien.ac.at>.

The use of such decompositions can speed up the solving of CSPs and also the answering of CQs significantly. In fact, in [1] a speed-up up to a factor of 2,500 was reported for the CQs studied there. Structural decompositions are therefore already being used in commercial products and research prototypes, both in the CSP area as well as in database systems [1, 4, 5, 27, 33]. However, previous decomposition algorithms are limited in that they fail to find optimal decompositions (i.e., decompositions of minimal width) even for low widths. This is also the case for various GHD computation methods proposed in [14, 31, 38]. The overall aim of our work is therefore to advance the art of computing hypergraph decompositions and to make the use of GHDs for solving CSPs applicable to a significantly wider range of CSP instances than previous methods. More specifically, we derive the following research goals:

Main Goal: Provide major improvements for computing hypergraph decompositions.

As part of this main goal, we define in particular:

Sub-goal 1: Design novel parallel algorithms for structural decompositions, in particular GHDs, and

Sub-goal 2: Put all this to work, by implementing and extensively evaluating these improvements.

Note that, apart from GHDs, there are also other types of hypergraph decompositions (see [18] for a comparison), notably tree decompositions (TDs) [37], hypertree decompositions (HDs) [19], and fractional hypertree decompositions (FHDs) [25]. TDs are the oldest and most intensively studied form of these decomposition methods, both in terms of their

efficient computation (see e.g. [7, 29]), the potential for parallel algorithms (see e.g. [32]) and their application to a wide range of problems – including constraint solving [36]. However, compared with the other types of decompositions mentioned above, TDs have a serious drawback in the context of CSP evaluation and CQ answering: CSP and CQ algorithms based on any of these decompositions essentially run in time $O(n^k)$, where n is the size of the problem instance and k is the width of the decomposition used. However, if we have relations of arity α , then the treewidth may be up to a factor α worse than the width notions based on the other decomposition methods. This is why HDs, GHDs, and FHDs are the better choice for CSPs and CQs.

There exist systems for computing HDs [14, 23, 38], GHDs [14, 31, 38], and FHDs [13]. However, to the best of our knowledge, none of them makes use of parallelism. In theory, HD-computation is easiest among the three. Indeed, the check problem (i.e., the problem of checking if a decomposition of some fixed width exists and, in the positive case, computing such a decomposition) is tractable for HDs [19] but intractable for GHDs and FHDs even for width = 2 [15, 20]. Nevertheless, despite this tractability result, HD-computation has turned out to be computationally expensive in practice. And parallelisation of the computation is tricky in this case since, in contrast to GHDs and FHDs, HDs are based on a *rooted* tree. This makes it impossible to reuse the ideas of the parallel GHD computation applied here, which recursively splits the task of computing the tree underlying a GHD into subtrees which then have to be re-rooted appropriately when they are stitched together. FHDs are computationally yet more expensive than GHDs. So GHDs are a good middle ground among these 3 types of decompositions.

As a first step in pursuing the first goal, we aim at *generally applicable* simplifications of hypergraphs to speed up the decomposition of hypergraphs. Here, “general applicability” means that these simplifications can be incorporated into any decomposition algorithms such as the ones presented in [13, 14] and also earlier work such as [23]. Moreover, we aim at heuristics for guiding the decomposition algorithms to explore more promising parts of the big search space first.

However, it will turn out that these simplifications and heuristics are not sufficient to overcome a principal shortcoming of existing decomposition algorithms, namely their sequential nature. Modern computing devices consist of multi-core architectures, and we can observe that single-core performance has mostly stagnated since the mid-2000s. So to produce programs which run optimally on modern machines, one must find a way of designing them to run efficiently in parallel. However, utilising multi-core systems is a non-trivial task, which poses several challenges. In our design of parallel GHD-algorithms, we focus on three key issues:

- i minimising synchronisation delay as much as possible,
- ii finding a way to partition the search space equally among CPUs, and thus utilising the resources optimally,
- iii supporting efficient backtracking, a key element of all structural decomposition algorithms presented so far.

In order to evaluate our algorithmic improvements and our new parallel GHD-algorithms, we have implemented them and tested them on the publicly available HyperBench benchmark mentioned above. For our implementation, we decided to use the programming language Go proposed by Google [10], which is based on the classical Communication Sequential Processes pattern by [28], since it reduces the need for explicit synchronisation.

To summarise, the **main results** of this work are as follows:

- We have developed three parallel algorithms for computing GHDs, where the first two are loosely based on the *balanced separator* method from [3, 14]. As has been mentioned above, none of the previous systems for computing HDs, GHDs, or FHDs makes use of parallelism. Our parallel approach has opened the way for a *hybrid approach*, which combines the strengths of parallel and sequential algorithms. This hybrid approach ultimately proved to be the best.
- In addition to designing parallel algorithms, we propose several algorithmic improvements such as applying multiple pre-processing steps on the input hypergraphs and using various heuristics to guide the search for a decomposition. While most of the pre-processing steps have already been used before, their combination and, in particular, a proof that their exhaustive application yields a unique normal form (up to isomorphism) is new. Moreover, for the hybrid approach, we have explored when to best switch from one approach to the other.
- We have implemented the parallel algorithms together with all algorithmic improvements and heuristics presented here. The source code of the program is available under <https://github.com/cem-okulmus/BalancedGo>. With our new algorithms and their implementation, dramatically more instances from HyperBench could be solved compared with previous algorithms. More specifically, we could extend the number of hypergraphs with exact *ghw* known by over 50%. In total, this means that for over 75% of all instances of HyperBench, the exact *ghw* is now known. If we leave aside the randomly-generated CSPs, and focus on the those from real world applications, we can show an increase of close to 100%, thus almost doubling the number of instances solved.

Our work therefore makes it possible to compute GHDs efficiently on modern machines for a wide range of CSPs. It enables the fast recognition of low widths for many instances encountered in practice (as represented by HyperBench) and thus lays the foundation for more systems and applications in evaluating CSPs and CQs based on their structural properties.

The remainder of this paper is structured as follows: In Section 2, we provide the needed terminology and recall previous approaches. In Section 3, we present our general algorithmic improvements. This is followed by a description of our parallelisation strategy in Section 4. Experimental evaluations are presented in Section 5. In Section 6, we summarise our main results and highlight directions for future work. This paper is an enhanced and extended version of work presented at IJCAI-PRICAI 2020 [21].

2 Preliminaries

CSPs & hypergraphs A *constraint satisfaction problem* (CSP) P is a set of *constraints* (S_i, R_i) with $1 \leq i \leq m$, where each $S_i = \{s_0, \dots, s_n\}$ is a set of variables and R_i a constraint relation which contains tuples of size n using values from a domain D . A solution to P is a mapping of variables to values from the domain D , such that for each constraint we map the variables to some tuple in its constraint relation.

A *hypergraph* H is a tuple $(V(H), E(H))$, consisting of a set of vertices $V(H)$ and a set of hyperedges (synonymously, simply referred to as “edges”) $E(H) \subseteq 2^{V(H)}$,

where the notation $2^{V(H)}$ signifies the power set over $V(H)$. To get the hypergraph of a CSP P , we consider $V(H)$ to be the set of all variables in P , to be precise $\bigcup_i S_i$, and each S_i to be one hyperedge. Here, we disregard the constraint relations, as they contain no additional structural information.

Recall that solving a CSP corresponds to model checking a first-order formula Φ (representing the constraints S_i) over a finite structure (made up by the relations R_i) such that the only connectives allowed in Φ are \exists and \wedge , whereas \forall , \vee , and \neg are disallowed. Hence, formally, CSP solving is equivalent to answering conjunctive queries (CQs) in the database world [30, 35]. In the sequel, we will mainly concentrate on CSPs with the understanding that all our results equally apply to CQs.

The *intersection size* of a hypergraph H is defined as the minimum integer c , such that for any two edges $e_1, e_2 \in E(H)$, $e_1 \cap e_2 \leq c$. A class \mathcal{C} of hypergraphs has the *bounded intersection property* (BIP), if there exists a constant c such that every hypergraph $H \in \mathcal{C}$ has intersection size $\leq c$.

We are frequently dealing with sets of sets of vertices (e.g., sets of edges). For $S \subseteq 2^{V(H)}$, we write $\bigcup S$ and $\bigcap S$ as a short-hand for taking the union or intersection, respectively, of this set of sets of vertices, i.e., for $S = \{s_1, \dots, s_\ell\}$, we have $\bigcup S = \bigcup_{i=1}^\ell s_i$ and $\bigcap S = \bigcap_{i=1}^\ell s_i$. For a set S of edges, we will alternatively also write $V(S)$ to denote the vertices contained in any of the edges in S . That is, we have $V(S) = \bigcup S$.

Decompositions A *generalized hypertree decomposition* (GHD) of a hypergraph $H = (V(H), E(H))$ is a tuple $\langle T, \chi, \lambda \rangle$, where $T = (N, E(T))$ is a tree, and χ and λ are labelling functions, which map to each node $n \in N$ two sets, $\chi(n) \subseteq V(H)$ and $\lambda(n) \subseteq E(H)$. For a node n we call $\chi(n)$ the *bag*, and $\lambda(n)$ the *edge cover* of n . We denote with $B(\lambda(n))$ the set $\{v \in V(H) \mid v \in e \text{ for some } e \in \lambda(n)\}$, i.e., the set of vertices “covered” by $\lambda(n)$. The functions χ and λ have to satisfy the following conditions:

1. For each $e \in E(H)$, there is a node $n \in N$ s.t. $e \subseteq \chi(n)$.
2. For each vertex $v \in V(H)$, $\{n \in N \mid v \in \chi(n)\}$ is a connected subtree of T .
3. For each node $n \in N$, we have that $\chi(n) \subseteq B(\lambda(n))$.

The second condition is also referred to as the *connectedness condition*. The *width* of a GHD is defined as $\max\{|\lambda(n)| \mid n \in N\}$. The generalized hypertree width (ghw) of a hypergraph is the smallest width of any of its GHDs. Deciding if $\text{ghw}(H) \leq k$ for a hypergraph H and fixed k is NP-complete, as one needs to consider exponentially many possible choices for the bag $\chi(n)$ for a given edge cover $\lambda(n)$.

It was shown in [14] that for any class of hypergraphs enjoying the BIP, one only needs to consider a polynomial set of subsets of hyperedges (called *subedges*) to compute their ghw. This fact will be explained in more detail in Section 2.2.

Example 1 An example of a hypergraph is shown in Fig. 1, as well as a GHD of this hypergraph. We can see that no λ -label uses more than two hyperedges, and thus this GHD has width 2, and the ghw of the hypergraph is also ≤ 2 . In fact, the hypergraph contains alpha cycles [12], e.g., $\{e_2, e_3, e_4, e_5\}$. Hence, we also know its ghw must be > 1 . Taken together, its ghw is therefore exactly 2.

Components & separators Consider a set of vertices $W \subseteq V(H)$. A set of edges $C \subseteq E(H)$ is $[W]$ -connected if for any two distinct edges $e, e' \in C$ there exists a sequence of vertices

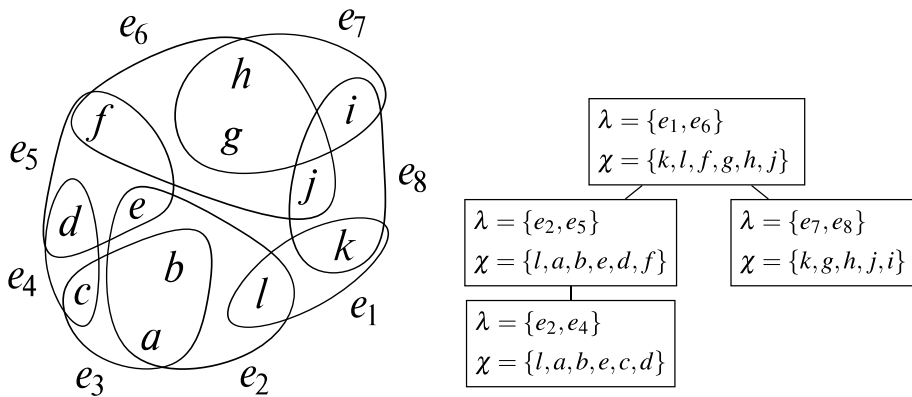


Fig. 1 An example hypergraph, where the vertices are represented by letters, with explicit edge names, together with a GHD of width 2

v_1, \dots, v_h and a sequence of edges e_0, \dots, e_h ($h \geq 1$) with $e_0 = e$ and $e_h = e'$ such that $v_i \in e_{i-1} \cap e_i$ and $v_i \notin W$ for each $i \in \{1, \dots, h\}$. In other words, there is a path from e to e' which only goes through vertices outside W . A set $C \subseteq E(H)$ is a $[W]$ -component, if C is maximal $[W]$ -connected. For a set of edges $S \subseteq E(H)$, we say that C is “[S]-connected” or an “[S]-component” as a short-cut for C is “[W]-connected” or a “[W]-component”, respectively, with $W = \bigcup S$. We also call S a *separator* in this context. The *size of an* [S]-component C is simply its cardinality. For a hypergraph H and a set of edges $S \subseteq E(H)$, we say that S is a *balanced separator* if all [S]-components of H have size $\leq \frac{|E(H)|}{2}$.

Example 2 An example for a separator that generates multiple connected components can be seen in Fig. 2. The separator S consists of two hyperedges e_2, e_6 , marked by 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.

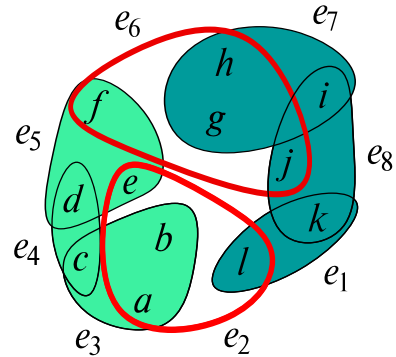
It was shown in [2] that, for every GHD $\langle T, \chi, \lambda \rangle$ of a hypergraph H , there exists a node $n \in N$ such that $\lambda(n)$ is a balanced separator of H . This property can be used when searching for a GHD of size k of H , as we shall recall in Section 2.2 below.

2.1 Computing hypertree decompositions (HDs)

We briefly recall the basic principles of the det- k -decomp program from [23] for computing Hypertree Decompositions (HDs), which was the first implementation of the original HD algorithm from [19]. HDs are GHDs with an additional condition to make their computation tractable in a way explained next.

For fixed $k \geq 1$, det- k -decomp tries to construct an HD of a hypergraph H in a top-down manner. It thus maintains a set C of edges, which is initialised to $C := E(H)$. For a node n in the HD (initially, this is the root of the HD), it “guesses” an edge cover $\lambda(n)$, i.e., $\lambda(n) \subseteq E(H)$ and $|\lambda(n)| \leq k$. For fixed k , there are only polynomially many possible values $\lambda(n)$. det- k -decomp then proceeds by determining all $[\lambda(n)]$ -components C_i with $C_i \subseteq C$.

Fig. 2 Connected components and their respective separator, visually marked



The additional condition imposed on HDs (compared with GHDs) restricts the possible choices for $\chi(n)$ and thus guarantees that the $[\lambda(n)]$ -components inside C and the $[\chi(n)]$ -components inside C coincide. This is the crucial property for ensuring polynomial time complexity of HD-computation – at the price of possibly missing GHDs with a lower width.

Now let C_1, \dots, C_ℓ denote the $[\lambda(n)]$ -components with $C_i \subseteq C$. By the maximality of components, these sets $C_i \subseteq E(H)$ are pairwise disjoint. Moreover, it was shown in [19] that if H has an HD of width $\leq k$, then it also has an HD of width $\leq k$ such that the edges in each C_i are “covered” in different subtrees below n . More precisely, this means that n has ℓ child nodes n_1, \dots, n_ℓ , such that for every i and every $e \in C_i$, there exists a node n_e in the subtree rooted at n_i with $e \subseteq \chi(n_e)$. Hence, $\text{det-}k\text{-decomp}$ recursively searches for an HD of the hypergraphs H_i with $E(H_i) = C_i$ and $V(H_i) = \bigcup C_i$ with the slight extra feature that also edges from $E(H) \setminus C_i$ are allowed to be used in the λ -labels of these HDs.

2.2 Computing GHDs

It was shown in [15] that, even for fixed $k = 2$, deciding if $\text{ghw}(H) \leq k$ holds for a hypergraph H is NP-complete. However, it was also shown there that if a class of hypergraphs satisfies the BIP, then the problem becomes tractable. The main reason for the NP-completeness in the general case is that, for a given edge cover $\lambda(n)$, there can be exponentially many bags $\chi(n)$ satisfying condition 3 of GHDs, i.e., $\chi(n) \subseteq B(\lambda(n))$. In other words, to have a sound and complete procedure to check if a given hypergraph has ghw at most k , one would need to check exponentially many possible bags for any given edge cover.

Now let $\lambda(n) = \{e_{i_1}, \dots, e_{i_\ell}\}$ with $\ell \leq k$. Of course, if we restrict each e_{i_j} to the sub-edge $e'_{i_j} = e_{i_j} \cap \chi(n)$ and define $\lambda'(n) = \{e'_{i_1}, \dots, e'_{i_\ell}\}$, then we get $\chi(n) = B(\lambda'(n))$. The key to the tractability shown in [15] in case of the BIP (i.e., the intersection of any two distinct edges is bounded by a constant b) is twofold: first, it is easy to see that w.l.o.g., we may restrict the search for a GHD of desired width k to so-called “bag-maximal” GHDs. That is, for any node n , it is impossible to add another vertex to $\chi(n)$ without

violating a condition from the definition of GHDs. And second, it is then shown in [15] for bag-maximal GHDs, that each e'_{ij} is either equal to e_{ij} or a subset of e_{ij} with $|e'_{ij}| \leq k \cdot b$. Hence, there are only polynomially many choices of subedges e'_{ij} and also of $\chi(n)$. More precisely, for a given edge e , the set of subedges to consider is defined as follows:

$$f_e(H, k) = \bigcup_{e_1, \dots, e_j \in (E(H) \setminus \{e\}), j \leq k} 2^{(e \cap (e_1 \cup \dots \cup e_j))} \quad (1)$$

In [14], this property was used to design a program for GHD computation as a straightforward extension of *det- k -decomp* by adding the *polynomially many* subedges $f_e(H, k)$ for all $e \in E(H)$ to $E(H)$. In the hypergraph extended in this way, we can thus be sure that $\lambda(n)$ can always be replaced by $\lambda'(n)$ with $\chi(n) = B(\lambda'(n))$.

In [14], yet another GHD algorithm was presented. It is based on the use of *balanced separators* and extends ideas from [3]. The motivation of this approach comes from the observation that there is no useful upper bound on the size of the subproblems that have to be solved by the recursive calls of the *det- k -decomp* algorithm. In fact, for some node n with corresponding component C , let C_1, \dots, C_ℓ denote the $[\lambda(n)]$ -components with $C_i \subseteq C$. Then there may exist an i such that C_i is “almost” as big as C . In other words, in the worst case, the recursion depth of *det- k -decomp* may be linear in the number of edges.

The Balanced Separator approach from [14] uses the fact that every GHD must contain a node whose λ -label is a balanced separator. Hence, in each recursive decomposition step for some subset E' of the edges of H , the algorithm “guesses” a node n' such that $\lambda(n')$ is a balanced separator of the hypergraph with edges E' . Of course, this node n' is not necessarily a child node n_i of the current node n but may lie somewhere inside the subtree T_i below n . However, since GHDs can be arbitrarily rooted, one may first compute this subtree T_i with n' as the root and with n_i as a leaf node. This subtree is then (when returning from the recursion) connected to node n by rerooting T_i at n_i and turning n_i into a child node of n . The definition of balanced separators guarantees that the recursion depth is logarithmically bounded. This makes the Balanced Separator algorithm a good starting point for our parallel algorithm to be presented in Section 4.

3 Algorithmic improvements

In this section, we present several algorithmic improvements of decomposition algorithms. We start with some simplifications of hypergraphs, which can be applied as a preprocessing step for any hypergraph decomposition algorithm, i.e., they are not restricted to the GHD algorithms discussed here. We shall then also mention further algorithmic improvements which are specific to the GHD algorithms presented in this paper. We note that, while the GHD-specific algorithmic improvements are new, the simplifications mentioned below have already been used before and/or are quite straightforward. We prove that their exhaustive application to an arbitrary hypergraph yields a unique normal form up to isomorphism. For the sake of completeness, we also prove the correctness and polynomial time complexity of their application.

3.1 Hypergraph preprocessing

An important step to speed up decomposition algorithms is the simplification of the input hypergraph. Before we formally present such a simplification, we observe that we may restrict ourselves to *connected* hypergraphs, formally those having only a single $[\emptyset]$ -component, since a GHD of a hypergraph consisting of several connected components can be obtained by combining the GHDs of each connected component in an “arbitrary” way, e.g., appending the root of one GHD as a child of an arbitrarily chosen node of another GHD. This can never violate the connectedness condition, since the GHDs of different components have no vertices in common. It is easy to verify that the simplifications proposed below never make a connected hypergraph disconnected. Hence, splitting a hypergraph into its connected components can be done upfront, once and for all. After that, we are exclusively concerned with connected hypergraphs. Given a (connected) hypergraph $H = (V(H), E(H))$, we thus propose the exhaustive application of the following reduction rules in a don’t-care non-deterministic fashion:

The so-called GYO reduction was introduced in [24, 42] to test the acyclicity of a hypergraph. It consists of the Rules 1 and 2 recalled below:

- Rule 1.** Suppose that H contains a vertex v that only occurs in a single edge e . Then we may delete v from e and thus from $V(H)$ altogether.
- Rule 2.** Suppose that H contains two edges e_1, e_2 , such that $e_1 \subseteq e_2$. Then we may delete e_1 from $E(H)$.

The next reduction of hypergraphs makes use of the notion of *types* of vertices. Here the *type* of a vertex v is defined as the set of edges e which contain v . We thus define Rule 3 as follows:

- Rule 3.** Suppose that H contains vertices v_1, v_2 of the same *type*. Then we may delete v_2 from $V(H)$ and thus from all edges containing v_2 .

The next reduction rule considered here uses the notion of *hinges*. In [26], hinge decompositions were introduced to help split CSPs into smaller subproblems. In [17], the combination of hinge decompositions and hypertree decompositions was studied. We also make use of hinge decompositions as part of our preprocessing. More specifically, we define the following reduction rule:

- Rule 4.** Let $e \in E(H)$ and let $\mathcal{C} = \{C_1, \dots, C_\ell\}$ with $\ell \geq 2$ denote the $[e]$ -components of H . Then we may split H into hypergraphs $H_1 = (V(H_1), E(H_1)), \dots, H_\ell = (V(H_\ell), E(H_\ell))$ with $H(E_i) = C_i \cup \{e\}$ and $V(H_i) = \bigcup E(H_i)$ for each i .

The above simplifications (above all the splitting into smaller hypergraphs via Rule 4) may produce a hypergraph that is so small that the construction of a GHD of width $\leq k$ for given $k \geq 1$ becomes trivial. The following rule allows us to eliminate such trivial cases:

- Rule 5.** If $|E(H)| \leq k$, then H may be deleted. It has a trivial GHD consisting of a single node n with $\lambda(n) = E(H)$ and $\chi(n) = \bigcup E(H)$.

In Theorems 1 and 2 below, we state several crucial properties of the reductions. Most importantly, these reductions neither add nor lose solutions. Moreover, preprocessing a hypergraph with these rules can be done in polynomial time.

Note that, even though all Rules 1 – 5 are applied to a *single hypergraph*, the result in case of Rule 4 is a *set of hypergraphs*. Hence, strictly speaking, these rules form a rewrite system that transforms a set of hypergraphs into another set of hypergraphs, where the starting point is a singleton consisting of the initial hypergraph only. However, to keep the notation simple, we will concentrate on the effect of these rules on a single hypergraph with the understanding that application of one of these rules comes down to selecting an element from a set of hypergraphs and replacing this element by the hypergraph(s) according to the above definition of the rules.

Theorem 1 *Preprocessing an input hypergraph H via Rules 1 – 5 is sound. More precisely, let $\{H_1, \dots, H_m\}$ be the result of exhaustive application of Rules 1 – 5 to a hypergraph H . Then, for any $k \geq 1$, we have $\text{ghw}(H) \leq k$ if and only if, for every $i \in \{1, \dots, m\}$, $\text{ghw}(H_i) \leq k$ holds.*

As for the complexity, this transformation of H into $\{H_1, \dots, H_\ell\}$ is feasible in polynomial time. Moreover, any collection of GHDs of width $\leq k$ of H_1, \dots, H_ℓ can be transformed in polynomial time into a GHD of H of width $\leq k$.

Proof We split the proof in two main parts: first, we consider the complexity of exhaustive application of Rules 1 – 5 and then we prove the soundness of the rules. The polynomial-time complexity of constructing a GHD of H from GHDs of the resulting hypergraphs $\{H_1, \dots, H_m\}$ will be part of the correctness proof. \square

Complexity of exhaustive rule application Rules 1 – 3 have the effect that the size of H is strictly decreased by either deleting vertices or edges. Hence, there can be only linearly many applications of Rules 1 – 3 and each of these rule applications is clearly feasible in polynomial time. Likewise, Rule 5, which allows us to delete a non-empty hypergraph, can only be applied linearly often and any application of this rule is clearly feasible in polynomial time. Checking if Rule 4 is applicable and actually applying Rule 4 is also feasible in polynomial time. Hence, it only remains to show that the total number of applications of Rule 4 is polynomially bounded. To see this, we first of all make the following observation on the number of edges in each H_i : consider a single application of Rule 4 and suppose that, for some edge e , there are ℓ $[e]$ -components C_1, \dots, C_ℓ . These $[e]$ -components are pairwise disjoint and we have $C_i \subseteq E(H) \setminus \{e\}$ for each i . Hence, if $|E(H)| = n$ and $|C_i| = n_i$ with $n_i \geq 1$, then $n_1 + \dots + n_\ell \leq n - 1$ holds. Moreover, $|E(H_i)| = n_i + 1$, since we add e to each component. We claim that, in total, when applying Rules 1 – 4 exhaustively to a hypergraph H with $n \geq 3$ edges, there can be at most $2n - 3$ applications of Rule 4. Note that for $n = 1$ or $n = 2$, Rule 4 is not applicable at all.

We prove this claim by induction on n : if H has 3 edges, then an application of Rule 4 is only possible, if we find an edge e , such that there are 2 $[e]$ -components C_1, C_2 , each consisting of a single edge. Hence, such an application of Rule 4 produces two hypergraphs H_1, H_2 with 2 edges each, to which no further application of Rule 4 is possible. Hence, the total number of applications of Rule 4 is bounded by 1 and, for $n = 3$, we indeed have $1 \leq 6 - 3 \leq 2n - 3$.

For the induction step, suppose that the claim holds for any hypergraph with $\leq n - 1$ edges and suppose that H has n edges. Moreover, suppose that an application of Rule 4 for some edge e is possible with $\ell \geq 2$ $[e]$ -components C_1, \dots, C_ℓ and let $|C_i| = n_i$. Then H is split into ℓ hypergraphs H_1, \dots, H_ℓ with $|E(H_i)| = n_i + 1$. Note that applications of any

of the Rules 1 – 3 to the hypergraphs H_i can never increase the number of edges. These rules may thus be ignored and we may apply the induction hypothesis to each H_i . Hence, for every i , there are at most $2(n_i + 1) - 3 = 2n_i - 1$ applications of Rule 4 in total possible for H_i . Taking all the resulting hypergraphs H_1, \dots, H_ℓ together, the total number of applications of Rule 4 is therefore $\leq (2n_1 + \dots + 2n_\ell) - \ell$. Together with the inequalities $n_1 + \dots + n_\ell \leq n - 1$ and $\ell \geq 2$, and adding the initial application of Rule 4, we thus have, in total, $\leq 2(n - 1) - \ell + 1 = 2n - 2 - \ell + 1 \leq 2n - 2 - 2 + 1 = 2n - 3$ applications of Rule 4.

Soundness For the soundness of our reduction system, we have to prove the soundness of each single rule application. Likewise, for the polynomial-time complexity of constructing a GHD of H from the GHDs of the final hypergraph set $\{H_1, \dots, H_m\}$, it suffices to show that one can efficiently construct a GHD of the original hypergraph from the GHD(s) of the hypergraph(s) resulting from a single rule application. This is due to the fact that we have already shown above that the total number of rule applications is polynomially bounded. It thus suffices to prove the following claim:

Claim A *Let H be a hypergraph and suppose that H' is the result of a single application of one of the Rules 1 – 3 to H . Then $\text{ghw}(H) \leq k$ if and only if $\text{ghw}(H') \leq k$. Moreover, in the positive case, a GHD of H of width $\leq k$ can be constructed from a GHD of H' of width $\leq k$ in polynomial time.*

Likewise, suppose that H_1, \dots, H_ℓ is the result of a single application of Rule 4 to H . Then $\text{ghw}(H) \leq k$ if and only if, for every $i \in \{1, \dots, \ell\}$, $\text{ghw}(H_i) \leq k$ holds. Moreover, in the positive case, a GHD of H of width $\leq k$ can be constructed from GHDs of H_1, \dots, H_ℓ of width $\leq k$ in polynomial time.

Note that we have omitted Rule 5 in this claim, since both the soundness and the polynomial-time construction of a GHD of width $\leq k$ are trivial. The proof of Claim A is straightforward but lengthy due to the case distinction over the 4 remaining rules. It is therefore deferred to Section 3.3.

Note that the application of one rule may enable the application of another rule; so their combination may lead to a greater simplification compared to just any one rule alone. Now the question naturally arises if the order in which we apply the rules has an impact on the final result. We next show that exhaustive application of Rules 1 – 5 leads to a unique (up to isomorphism) result, even if they are applied in a don't-care non-deterministic fashion.

Theorem 2 *Transforming a given hypergraph with Rules 1 – 5 leads to a unique normal form. That is, let H be a hypergraph and let $\{H_1, \dots, H_m\}$ be the result of exhaustively applying Rules 1 – 5. Then $\{H_1, \dots, H_m\}$ is unique (up to isomorphism) no matter in which order the Rules 1 – 5 are applied.*

Proof Recall that, in Theorem 1, we have already shown that the rewrite system is terminating (actually, we have even shown that there are at most polynomially many rule applications). In order to show that the rewrite system guarantees a unique normal form (up to isomorphism), it is therefore sufficient to show that it is *locally confluent* [6]. That is, we have to prove the following property: Let \mathcal{H} be a set of hypergraphs and suppose that there are two possible ways of applying Rules 1 – 5 to (an element H of) \mathcal{H} , so that \mathcal{H} can be

Table 1 Overview of the complexity of the four methods considered for ordering hyperedges

Method of edge ordering	Runtime worst case complexity
Maximal cardinality search ordering	$O(E(H) ^2)$
Maximal separator ordering	$O(E(H) \cdot V(H) ^3)$
Vertex degree ordering	$O(E(H) ^2)$
Edge degree ordering	$O(E(H) ^2)$

transformed to either \mathcal{H}_1 or \mathcal{H}_2 . Then there exists a set of hypergraphs \mathcal{H} , such that both \mathcal{H}_1 and \mathcal{H}_2 can be transformed into \mathcal{H} by a sequence of applications of Rules 1 – 5. In the notation of [6], this property is succinctly presented as follows:

$$\mathcal{H}_1 \leftarrow \mathcal{H} \rightarrow \mathcal{H}_2 \Rightarrow \mathcal{H}_1 \downarrow \mathcal{H}_2$$

To prove this property, we have to consider all possible pairs (i, j) of applicable Rules i and j .

This case distinction is rather tedious (especially the cases where Rule 4 is involved) but not difficult. We thus defer the details to Section 3.4.

3.2 Finding balanced separators fast

It has already been observed in [23] that the ordering in which edges are considered is vital for finding an appropriate edge cover $\lambda(n)$ for the current node n in the decomposition fast. However, the ordering used in [23] for *det- k -decomp*, (which was called MCSO, i.e., maximal cardinality search ordering) turned out to be a poor fit for finding balanced separators. A natural alternative was to consider, for each edge e , all possible paths between vertices in the hypergraph H , and how much the length of these paths increases after removal of e . This provides a weight for each edge, based on which we can define the *maximal separator ordering*. In our tests, this proved to be a very effective heuristic. Unfortunately, computing the maximal separator ordering requires solving the all-pairs shortest path problem. Using the well-known Floyd-Warshall algorithm [16, 40] as a subroutine, this leads to a fairly high complexity – see Table 1 – which proved to be prohibitively expensive for practical instances. We thus explored two other, computationally simpler, heuristics, which order the edges in descending order of the following measures:

- The *vertex degree* of an edge e is defined as $\sum_{v \in e} \deg(v)$, where $\deg(v)$ denotes the degree of a vertex v , i.e., the number of edges containing v .
- The *edge degree* of an edge e is $|\{f : e \cap f \neq \emptyset\}|$, i.e., the number of edges e has a non-empty intersection with.

In our empirical evaluation, we found both of these to be useful compromises between speeding up the search for balanced separators and the complexity of computing the ordering itself, with the vertex degree ordering yielding the best results, i.e., compute $\lambda(n)$ by first trying to select edges with higher vertex degree.

Finding the next balanced separator Finding a balanced separator fast is important for the performance of our GHD algorithm, but it is not enough: if the balanced separator thus found does not lead to a successful GHD computation, we have to try another one. Hence, it is important to find the next balanced separator fast and to avoid trying the same balanced separator multiple times. The GHD algorithm based on balanced separators presented in [14] searches through all ℓ -tuples of edges (with $\ell \leq k$) to find the next balanced separator. The number of edge-combinations thus checked is $\sum_{i=1}^k \binom{N}{i}$, where N denotes the number of edges. Note that this number of edges is actually higher than in the input hypergraph due to the subedges that have to be added for the tractability of GHD computation (see Section 2.2). Before we explain our improvement, let us formally explain how subedges factor into the search. Let us assume that we are given an edge cover (e_1, \dots, e_k) , consisting of exactly k edges. Using the function $f_e(H, k)$ which generates the set of subedges to consider for any given edge e , defined in Section 2.2, we get the following set of edge combinations when factoring in all the relevant subedges:

$$\{(e'_1, \dots, e'_k) \mid e'_i \in \{e_i \cup f_{e_i}(H, k)\}, 1 \leq i \leq k\}$$

We note a significant source of redundancy in this set. If one only focuses on the combination of $l \leq k$ edges to intersect with e , it is possible that the same bags (when taking the union of their vertices) can be generated multiple times

We can address this by shifting our focus on the actual bags $\chi(n)$ generated from each $\lambda(n)$ thus computed. Therefore, we initially only look for balanced separators of size k , checking $\binom{N}{k}$ many initial choices of $\lambda(n)$. Only if a choice of $\lambda(n)$ and $\chi(n) = \bigcup \lambda(n)$ does not lead to a successful recursive call of the decomposition procedure, we also inspect subsets of $\chi(n)$ – strictly avoiding the computation of the same subset of $\chi(n)$ several times by inspecting different subedges of the original edge cover $\lambda(n)$. We thus also do not add subedges to the hypergraph upfront but only as they are needed as part of the backtracking when the original edge cover $\lambda(n)$ did not succeed. Separators consisting of fewer edges are implicitly considered by allowing also the empty set as a possible subedge.

Summary Our initial focus was to speed up existing decomposition algorithms via improvements as described above. However, even though these algorithmic improvements showed some positive effect, it turned out that a more fundamental change is needed. We have thus turned our attention to parallelisation, which will be the topic of Section 4. But first we present the missing parts of the proofs of Theorems 1 and 2 in Sections 3.3 and 3.4, respectively.

3.3 Completion of the proof of theorem 1

It remains to prove Claim A from the proof in Section 3.1.

Proof Proof of the Claim We prove the claim for each rule separately. It is convenient to treat $E(H)$ as a multiset, i.e., $E(H)$ may contain several “copies” of an edge. This simplifies the argumentation below, when the deletion of vertices may possibly make two edges identical. Note that, if at all, this only happens in intermediate steps, since Rule 2 above will later lead to the deletion of such copies anyway. \square

Rule 1 $H = (V(H), E(H))$ contains a vertex v that only occurs in a single edge e and we delete v from e and from $V(H)$ altogether. Let $e' = e \setminus \{v\}$. Then $H' = (V(H'), E(H'))$ with $V(H') = V(H) \setminus \{v\}$ and $E(H') = (E(H) \setminus \{e\}) \cup \{e'\}$. \Rightarrow : Let $\mathcal{D} = \langle T, \chi, \lambda \rangle$ be a GHD of H of width $\leq k$. We construct GHD $\mathcal{D}' = \langle T', \chi', \lambda' \rangle$ as follows: the tree structure T remains unchanged, i.e., we set $T' = T$. For every node n in the tree T' , we define $\lambda'(n)$ and $\chi'(n)$ as follows:

- If $e \in \lambda(n)$, then $\lambda'(n) = (\lambda(n) \setminus \{e\}) \cup \{e'\}$.
- If $v \in \chi(n)$, then $\chi'(n) = \chi(n) \setminus \{v\}$.
- For all other nodes n in T' , we set $\lambda'(n) = \lambda(n)$ and $\chi'(n) = \chi(n)$.

It is easy to verify that \mathcal{D}' is a GHD of H' . Moreover, the width clearly does not increase by this transformation. \Leftarrow : Let $\mathcal{D}' = \langle T', \chi', \lambda' \rangle$ be a GHD of H' of width $\leq k$. By the definition of GHDs, T' must contain at least one node n , such that $e' \subseteq \chi'(n)$. We arbitrarily choose one such node \hat{n} with $e' \subseteq \chi'(\hat{n})$. Then we construct GHD $\mathcal{D} = \langle T, \chi, \lambda \rangle$ as follows:

- T contains all nodes and edges from T' plus one additional leaf node n' which we append as a child node of \hat{n} .
- For n' , we set $\lambda(n') = \{e\}$ and $\chi(n') = e$.
- Let n be a node in T' with $e' \in \lambda'(n)$. Then we set $\lambda(n) = (\lambda'(n) \setminus \{e'\}) \cup \{e\}$ and we leave χ' unchanged, i.e., $\chi(n) = \chi'(n)$.
- For all other nodes n in T , we set $\lambda(n) = \lambda'(n)$ and $\chi(n) = \chi'(n)$.

Clearly, \mathcal{D} can be constructed from \mathcal{D}' in polynomial time. Moreover, it is easy to verify that \mathcal{D} is a GHD of H . In particular, the connectedness condition is not violated by the introduction of the new node n' into the tree, since vertex $v \in \chi(n')$ occurs nowhere else in \mathcal{D} and all other vertices in $\chi(n')$ are also contained in $\chi(\hat{n})$ for the parent node \hat{n} of n' . Moreover, the width clearly does not increase by this transformation since the new node n' has $|\lambda(n')| = 1$ and for all other λ -labels, the cardinality has been left unchanged.

Rule 2 Suppose that $H = (V(H), E(H))$ contains two edges e_1, e_2 , such that $e_1 \subseteq e_2$ and we delete e_1 from $E(H)$, i.e., $H' = (V(H'), E(H'))$ with $V(H') = V(H)$ and $E(H') = E(H) \setminus \{e_1\}$. \Rightarrow : Let $\mathcal{D} = \langle T, \chi, \lambda \rangle$ be a GHD of H of width $\leq k$. We construct GHD $\mathcal{D}' = \langle T', \chi', \lambda' \rangle$ as follows: the tree structure T remains unchanged, i.e., we set $T' = T$. For every node n in the tree T' , we define $\lambda'(n)$ and $\chi'(n)$ as follows:

- If $e_1 \in \lambda(n)$, then $\lambda'(n) = (\lambda(n) \setminus \{e_1\}) \cup \{e_2\}$.
- For all other nodes n in T' , we set $\lambda'(n) = \lambda(n)$.
- For all nodes n in T' , we set $\chi'(n) = \chi(n)$.

It is easy to verify that \mathcal{D}' is a GHD of H' and that the width does not increase by this transformation. \Leftarrow : Let $\mathcal{D}' = \langle T', \chi', \lambda' \rangle$ be a GHD of H' of width $\leq k$. It is easy to verify that then \mathcal{D}' is also a GHD of H . Indeed, we only need to verify that T' contains a node n with $e_1 \subseteq \chi'(n)$. By the definition of GHDs, there exists a node n in T' with $e_2 \subseteq \chi'(n)$. Hence, since we have $e_1 \subseteq e_2$, also $e_1 \subseteq \chi'(n)$ holds.

Rule 3 Suppose that $H = (V(H), E(H))$ contains two vertices v_1, v_2 which occur in precisely the same edges and we delete v_2 from all edges and thus from $V(H)$ altogether, i.e., $H' = (V(H'), E(H'))$ with $V(H') = V(H) \setminus \{v_2\}$ and $E(H') = \{e \setminus \{v_2\} \mid e \in E(H)\}$.

It is convenient to introduce the following notation: suppose that $E(H) = \{e_1, \dots, e_\ell\}$. Then we denote $E(H')$ as $E(H') = \{e'_1, \dots, e'_\ell\}$, where $e'_i = e_i \setminus \{v_2\}$. Of course, we have $e'_i = e_i$ whenever $v_2 \notin e_i$. Let $\mathcal{D} = \langle T, \chi, \lambda \rangle$ be a GHD of H of width $\leq k$. We construct GHD $\mathcal{D}' = \langle T', \chi', \lambda' \rangle$ as follows: the tree structure T remains unchanged, i.e., we set $T' = T$. For every node n in the tree T' , we define $\lambda'(n)$ and $\chi'(n)$ as follows:

- Suppose that $\lambda(n) = \{e_{i_1}, \dots, e_{i_j}\}$ for some $j \leq k$. Then we set $\lambda'(n) = \{e'_{i_1}, \dots, e'_{i_j}\}$.
- For all nodes n in T' , we set $\chi'(n) = \chi(n) \setminus \{v_2\}$.

It is easy to verify that \mathcal{D}' is a GHD of H' and that the width does not increase by this transformation. \Leftarrow : Let $\mathcal{D}' = \langle T', \chi', \lambda' \rangle$ be a GHD of H' of width $\leq k$. Then we construct GHD $\mathcal{D} = \langle T, \chi, \lambda \rangle$ as follows: the tree structure T' remains unchanged, i.e., we set $T = T'$. For every node n in the tree T , we define $\lambda(n)$ and $\chi(n)$ as follows:

- Suppose that $\lambda'(n) = \{e'_{i_1}, \dots, e'_{i_j}\}$ for some $j \leq k$. Then we set $\lambda(n) = \{e_{i_1}, \dots, e_{i_j}\}$.
- For all nodes n in T' with $v_1 \in \chi'(n)$, we set $\chi(n) = \chi'(n) \cup \{v_2\}$.
- For all other nodes n in T' , we set $\chi(n) = \chi'(n)$.

Clearly this transformation is feasible in polynomial time and it does not increase the width. In order to show that \mathcal{D} is indeed a GHD of H , there are two non-trivial parts, namely: (1) for every $e_\alpha \in E(H)$, there exists a node n in T with $e_\alpha \subseteq \chi(n)$ and (2) $\chi(n) \subseteq B(\lambda(n))$ holds for every node n even if we add vertex v_2 to the χ -label. These are the two places where we make use of the fact that v_1 and v_2 occur in precisely the same edges in $E(H)$.

For part (1), note that there exists a node n in T' (and hence in T), such that $e'_\alpha \subseteq \chi'(n)$. If $v_1 \notin \chi'(n)$, then $v_1 \notin e'_\alpha$ and, therefore $v_1 \notin e_\alpha$. Hence, (since v_1 and v_2 have the same type) also $v_2 \notin e_\alpha$. We thus have $e_\alpha = e'_\alpha$ and $e_\alpha \subseteq \chi(n) = \chi'(n)$. On the other hand, if $v_1 \in \chi'(n)$, then $v_2 \in \chi(n)$ by the above construction of \mathcal{D} . Hence, $e_\alpha \subseteq \chi(n)$ again holds, since $e_\alpha \subseteq e'_\alpha \cup \{v_2\}$.

For part (2), consider an arbitrary vertex $v \in \chi(n)$. We have to show that $v \in B(\lambda(n))$. First, suppose that $v \neq v_2$. Then we have $v \in \chi'(n) \subseteq B(\lambda'(n)) \subseteq B(\lambda(n))$. It remains to consider the case $v = v_2$. Then, by the above construction of \mathcal{D} , we have $v_1 \in \chi'(n)$. We observe the following chain of implications: $v_1 \in \chi'(n) \Rightarrow v_1 \in e'_\alpha$ for some $e'_\alpha \in \lambda'(n) \Rightarrow v_1 \in e_\alpha$ for some $e_\alpha \in \lambda(n) \Rightarrow$ (since v_1 and v_2 have the same type) $v_2 \in e_\alpha$ for some $e_\alpha \in \lambda(n)$. That is, $v \in B(\lambda(n))$ indeed holds.

Rule 4 Suppose that $H = (V(H), E(H))$ contains an edge e with $[e]$ -components C_1, \dots, C_ℓ with $\ell \geq 2$. Further, suppose that we apply Rule 4 to replace H by the hypergraphs $H_1 = (V(H_1), E(H_1)), \dots, H_\ell = (V(H_\ell), E(H_\ell))$ with $H(E_j) = C_j \cup \{e\}$ and $V(H_j) = \bigcup E(H_j)$ for each j .

\Rightarrow : Let $\mathcal{D} = \langle T, \chi, \lambda \rangle$ be a GHD of H of width $\leq k$. We construct GHDs $\mathcal{D}_j = \langle T_j, \chi_j, \lambda_j \rangle$ of each H_j as follows: by the definition of GHDs, there must be a node n in T such that $e \subseteq \chi(n)$ holds. We choose such a node n and, w.l.o.g., we may assume that n is the root of \mathcal{D} . Let $\{D_1, \dots, D_m\}$ denote the $[\chi(n)]$ -components of H . It was shown in [19], that \mathcal{D} can be transformed into a GHD $\mathcal{D}' = \langle T', \chi', \lambda' \rangle$, such that the root node n is left unchanged (i.e.,

in particular, we have $\chi(n) = \chi'(n)$ and $\lambda(n) = \lambda'(n)$ and n has m child nodes n_1, \dots, n_m , such that there is a one-to-one correspondence between these child nodes and the $[\chi'(n)]$ -components D_1, \dots, D_m in the following sense: for every edge $e_i \in D_i$, there exists a node n'_i in the subtree rooted at n_i in T' such that $e_i \subseteq \chi'(n'_i)$. Intuitively, this means that the subtrees rooted at each of the child nodes of n “cover” precisely one $[\chi'(n)]$ -component. We make the following crucial observations:

1. For every $[\chi'(n)]$ -component D_i , there exists a unique $[e]$ -component C_j , such that $D_i \subseteq C_j$. This is due to the fact that every $[\chi'(n)]$ -connected set of edges is also $[e]$ -connected, since $e \subseteq \chi'(n)$.
2. Let $D_0 = \{f \in E(H) \mid f \subseteq \chi'(n)\}$. Then $E(H)$ is partitioned into D_0, D_1, \dots, D_m . That is $D_0 \cup D_1 \cup \dots \cup D_m = E(H)$ and $D_i \cap D_j = \emptyset$ for every pair $i \neq j$ of indices. This property can be seen as follows: every edge $f \in E(H)$ with $f \not\subseteq \chi'(n)$ must be contained in some $[\chi'(n)]$ -component. Hence, $D_0 \cup D_1 \cup \dots \cup D_m = E(H)$ clearly holds. On the other hand, by the very definition of components, any two distinct $[\chi'(n)]$ -components D_i, D_j with $i \neq j$ and $i, j \geq 1$, are disjoint. Finally, also D_0 and any D_i with $i \geq 1$ are disjoint since an edge f with $f \subseteq \chi'(n)$ cannot be $[\chi'(n)]$ -connected with any other edge.

Then, for $j \in \{1, \dots, \ell\}$, we define a GHD $\mathcal{D}_j = \langle T_j, \chi_j, \lambda_j \rangle$ of H_j as follows:

- T_j is the subtree of T' consisting of the following nodes:
 - the root node n is contained in T_j ;
 - for every $i \in \{1, \dots, m\}$, if $D_i \subseteq C_j$, then all nodes in the subtree rooted at n_i are contained in T_j ;
 - no further nodes are contained in T_j .
- For every node \hat{n} in T_j , we set $\chi_j(\hat{n}) = \chi'(\hat{n}) \cap V(H_j)$.
- For every node \hat{n} in T_j , we distinguish two cases for defining $\lambda_j(\hat{n})$:
 - If $\lambda'(\hat{n}) \subseteq E(H_j)$ holds, then we set $\lambda_j(\hat{n}) = \lambda'(\hat{n})$.
 - If $\lambda'(\hat{n}) \not\subseteq E(H_j)$ holds, then $\delta = \lambda'(\hat{n}) \setminus E(H_j) \neq \emptyset$ holds. In this case, we set $\lambda_j(\hat{n}) = (\lambda'(\hat{n}) \setminus \delta) \cup \{e\}$.

It remains to verify that \mathcal{D}_j is indeed a GHD of width $\leq k$ of H_j .

1. Consider an arbitrary $f \in E(H_j)$. We have to show that there exists a node \hat{n} in T_j with $f \subseteq \chi_j(\hat{n})$. By the second observation above, we know that $f \in D_i$ for some $i \geq 0$. If $f \in D_0$, then $f \subseteq \chi_j(n)$ for the root node n holds and we are done.
On the other hand, if $f \in D_i$ for some $i \geq 1$, then there exists a node \hat{n} in the subtree of T' rooted at n_i with $f \subseteq \chi'(\hat{n})$. Moreover, since $D_i \cap D_0 = \emptyset$, we know that $f \not\subseteq e$ and, therefore, $f \in C_j$ holds. By $f \in C_j$ and $f \in D_i$, we have $D_i \subseteq C_j$. Hence, by our construction of \mathcal{D}_j , \hat{n} is a node in T_j . Moreover, $f \subseteq V(H_j)$ and $f \subseteq \chi'(\hat{n})$. Hence, we also have $f \subseteq \chi_j(\hat{n}) = \chi'(\hat{n}) \cap V(H_j)$.
2. Consider an arbitrary vertex $v \in V(H_j)$. We have to show that $\{\hat{n} \in N_j \mid v \in \chi_j(\hat{n})\}$ is a connected subtree of T_j , where N_j denotes the node set of T_j . Let \hat{n}_1 and \hat{n}_2 be two nodes in N_j with $v \in \chi_j(\hat{n}_1)$ and $v \in \chi_j(\hat{n}_2)$. Then also $v \in \chi'(\hat{n}_1)$ and $v \in \chi'(\hat{n}_2)$ hold. Hence, in the GHD \mathcal{D} , for every node \hat{n} on the path between \hat{n}_1 and \hat{n}_2 , we have $v \in \chi'(\hat{n})$. Hence, every such node \hat{n} also satisfies $v \in \chi_j(\hat{n})$ by the definition $\chi_j(\hat{n}) = \chi'(\hat{n}) \cap V(H_j)$.

3. Consider an arbitrary node \hat{n} in T_j . We have to show that $\chi_j(\hat{n}) \subseteq B(\lambda_j(\hat{n}))$ holds. We distinguish the two cases from the definition of $\lambda_j(\hat{n})$:
 - If $\lambda'(\hat{n}) \subseteq E(H_j)$ holds, then we have $\lambda_j(\hat{n}) = \lambda'(\hat{n})$. Hence, from the property $\chi'(\hat{n}) \subseteq B(\lambda'(\hat{n}))$ for the GHD \mathcal{D} and $\chi_j(\hat{n}) \subseteq \chi'(\hat{n})$ it follows immediately that $\chi_j(\hat{n}) \subseteq B(\lambda_j(\hat{n}))$ holds.
 - Now suppose that $\lambda'(\hat{n}) \not\subseteq E(H_j)$ holds and let $\delta = \lambda'(\hat{n}) \setminus E(H_j) \neq \emptyset$. In this case, we have $\lambda_j(\hat{n}) = (\lambda'(\hat{n}) \setminus \delta) \cup \{e\}$. By $\chi_j(\hat{n}) \subseteq V(H_j)$, in order to prove $\chi_j(\hat{n}) \subseteq B(\lambda_j(\hat{n}))$, it suffices to show that $B(\lambda_j(\hat{n})) \supseteq B(\lambda'(\hat{n})) \cap V(H_j)$. To this end, it actually suffices to show that every $f' \in \delta$ has the property $f' \cap V(H_j) \subseteq e$:
 By $f' \in \delta$, we have $f' \in C_{j'}$ for some $j' \neq j$. Hence, for every $f \in C_j$, we have $f' \cap f \subseteq e$ by the definition of $[e]$ -components. Moreover, of course, also $f' \cap e \subseteq e$ holds. Hence, we indeed have $f' \cap V(H_j) \subseteq e$.
4. Finally, the width of \mathcal{D}_j is clearly $\leq k$ since $\lambda_j(\hat{n})$ is either equal to $\lambda'(\hat{n})$ or we add e but only after subtracting a non-empty set δ from $\lambda'(\hat{n})$.

\Leftarrow : For $j \in \{1, \dots, \ell\}$, let $\mathcal{D}_j = \langle T_j, \chi_j, \lambda_j \rangle$ be a GHD of H_j of width $\leq k$. By the definition of GHDs and by the fact that $e \in E(H_j)$ holds for every j , there exists a node n_j in T_j with $e \subseteq \chi_j(n_j)$. W.l.o.g., we may assume that n_j is the root of T_j . Then we construct GHD $\mathcal{D} = \langle T, \chi, \lambda \rangle$ as follows:

- The tree structure T is obtained by introducing a new node n as the root of T , whose child nodes are n_1, \dots, n_ℓ and each tree T_j becomes the subtree of T rooted at n_j .
- For the root node n , we set $\chi(n) = e$ and $\lambda(n) = \{e\}$.
- For any other node \hat{n} of T , we have that \hat{n} comes from exactly one of the trees T_j . We thus set $\chi(\hat{n}) = \chi_j(\hat{n})$ and $\lambda(\hat{n}) = \lambda_j(\hat{n})$.

Clearly, \mathcal{D} can be constructed in polynomial time from the GHDs $\mathcal{D}_1, \dots, \mathcal{D}_\ell$. Moreover, the width of \mathcal{D} is obviously bounded by the maximum width over the GHDs \mathcal{D}_i . It remains to verify that \mathcal{D} is indeed a GHD of H .

1. Consider an arbitrary $f \in E(H)$. We have to show that there is a node \hat{n} in T , s.t. $f \subseteq \chi(\hat{n})$. By the definition of $[e]$ -components, we either have $f \in C_i$ for some i or $f \subseteq e$. If $f \in C_i$, then there exists a node \hat{n} in the subtree rooted at n_i with $\chi(\hat{n}) = \chi_i(\hat{n}) \supseteq f$. If $f \subseteq e$, then we have $f \subseteq \chi(n)$.
2. Consider an arbitrary vertex $v \in V(H)$. We have to show that $\{\hat{n} \in N \mid v \in \chi(\hat{n})\}$ is a connected subtree of T , where N denotes the node set of T . Let $v \in \chi(\hat{n}_1)$ and $v \in \chi(\hat{n}_2)$ for two nodes \hat{n}_1 and \hat{n}_2 in N and let \hat{n} be on the path between \hat{n}_1 and \hat{n}_2 . If both nodes are in some subtree T_i of T , then the connectedness condition carries over from \mathcal{D}_i to \mathcal{D} . If one of the nodes \hat{n}_1 and \hat{n}_2 is the root n of T , say $n = \hat{n}_1$, then $v \in e$. Moreover, we have $e \subseteq \chi(n_i)$ by our construction of \mathcal{D} . Hence, we may again use the connectedness condition on \mathcal{D}_i to conclude that $v \in \chi(\hat{n})$ for every node \hat{n} along the path between \hat{n}_1 and \hat{n}_2 . Finally, suppose that \hat{n}_1 and \hat{n}_2 are in different subtrees T_i and T_j . Then $v \in V(H_i) \cap V(H_j)$ holds and, therefore, $v \in e$ by the construction of H_i and H_j via different $[e]$ -components. Hence, we are essentially back to the previous case. That is, we have $v \in \chi(\hat{n})$ for every node \hat{n} along the path from n to \hat{n}_1 and for every node \hat{n} along the path from n to \hat{n}_2 . Together with $v \in \chi(n)$, we may thus conclude that $v \in \chi(\hat{n})$ indeed holds for every node \hat{n} along the path between \hat{n}_1 and \hat{n}_2 .

3. Consider an arbitrary node \hat{n} in T . We have to show that $\chi(\hat{n}) \subseteq B(\lambda(\hat{n}))$. Clearly, all nodes in a subtree T_i inherit this property from the GHD \mathcal{D}_i and also the root node n satisfies this condition by our definition of $\chi(n)$ and $\lambda(n)$.

3.4 Completion of the proof of theorem 2

We now make a case distinction over all possible pairs (i, j) of Rules i and j applicable to some hypergraphs $H_i, H_j \in \mathcal{H}$ and exhibit a concrete hypergraph set \mathcal{H}^* that can be obtained from \mathcal{H} no matter if we first apply Rule i to H_i or Rule j to H_j . Note that we only need to consider the cases $i \leq j$, since the cases $i > j$ are thus covered by symmetry. Moreover, the only non-trivial case is that both Rules i and j are applied to the same hypergraph, i.e., $H_i = H_j = H$ for some hypergraph $H \in \mathcal{H}$.

“(i,5)”: local confluence is immediate for any combination of Rule 5 with another rule. Let $H \in \mathcal{H}$ with $|E(H)| \leq k$ and suppose that some other rule is also applicable to H . Then the desired hypergraph set \mathcal{H}^* is $\mathcal{H}^* = \mathcal{H} \setminus \{H\}$. Clearly, \mathcal{H}^* is the result of applying Rule 5 to $H \in \mathcal{H}$ and no further rule application is required in this case. Now suppose that another rule is applied first to H : Rules 1, 2, and 3 allow us to delete a vertex or an edge. In particular, the number of edges of the resulting hypergraph is still $\leq k$ and we may apply Rule 5 afterwards to get \mathcal{H}^* . Now suppose that Rule 4 is applicable to H . This means that we may replace H by several hypergraphs H_1, \dots, H_ℓ with $\ell \geq 2$. However, all these hypergraphs satisfy $|E(H_i)| < |E(H)| \leq k$. Hence, we may apply Rule 5 to each of them and delete all of the hypergraphs H_1, \dots, H_ℓ so that we again end up with \mathcal{H}^* .

“(1,1)”: Suppose that two applications of Rule 1 to some hypergraph $H \in \mathcal{H}$ are possible. That is, H contains a vertex v_1 that only occurs in a single edge e_1 and a vertex v_2 that only occurs in a single edge e_2 with $v_1 \neq v_2$. Note that, after deleting v_1 from $V(H)$, v_2 still occurs in a single edge e_2 . Likewise, after deleting v_2 from $V(H)$, v_1 still occurs in a single edge e_1 . Hence, \mathcal{H}^* is obtained by replacing H in \mathcal{H} by H' , which results from deleting both v_1 and v_2 from $V(H)$.

“(1,2)”: Suppose that an application of Rule 1 and an application of Rule 2 to the same hypergraph $H \in \mathcal{H}$ are possible. That is, H contains a vertex v that only occurs in a single edge e and H contains edges e_1, e_2 with $e_1 \subseteq e_2$. Hence, on one hand, we may delete v from H by Rule 1 and, on the other hand, we may delete e_1 from H by Rule 2. Note that $e_1 \neq e$, i.e., v cannot occur in e_1 since we are assuming that v occurs in a single edge and $e_1 \subseteq e_2$. Hence, after deleting v from $V(H)$, deletion of e_1 via Rule 2 is still possible, since we still have $e_1 \subseteq e_2$ and also $e_1 \subseteq (e_2 \setminus \{v\})$ (the latter relationship is relevant if $e = e_2$ and we actually delete v from e_2). Likewise, v still occurs in a single edge e after deleting e_1 via Rule 2. Hence, \mathcal{H}^* is obtained by replacing H in \mathcal{H} by H' , which results from deleting both v from $V(H)$ and e_1 from $E(H)$.

“(1,3)”: Suppose that an application of Rule 1 and an application of Rule 3 to the same hypergraph $H \in \mathcal{H}$ are possible. That is, H contains a vertex v that only occurs in a single edge e and H contains vertices v_1, v_2 of the same type, i.e., they occur in the same edges. If v is different from v_1 and v_2 , then we transform H into H' by deleting v and v_2 from H . If $v = v_2$, then Rule 1 and Rule 3 are simply two different ways of deleting node v from $V(H)$. Hence, the only interesting case remaining is that $v = v_1$ holds. In this case, also v_2 occurs

in edge e only, since we are assuming that v_1, v_2 are of the same type. Hence, \mathcal{H} is obtained by replacing H by the hypergraph H' which results from deleting both v_1 and v_2 from $V(H)$: if we first delete v_1 via Rule 1 then we may delete v_2 afterwards also via Rule 1. Conversely, if we first delete v_2 via Rule 3, then Rule 1 is still applicable to v_1 and we may thus delete it afterwards.

“(1,4)”: Suppose that an application of Rule 1 and an application of Rule 4 to the same hypergraph $H \in \mathcal{H}$ are possible. That is, H contains a vertex v_1 that only occurs in a single edge e_1 and H contains an edge e such that H has $[e]$ -components $\mathcal{C} = \{C_1, \dots, C_\ell\}$ with $\ell \geq 2$. Let $e'_1 = e_1 \setminus \{v_1\}$. Case 1. Suppose $e \neq e_1$. We have $e_1 \not\subseteq e$ since v_1 only occurs in e_1 . Hence, e_1 is contained in some $[e]$ -component C_i . We distinguish two subcases. Case 1.1. Suppose that $(e_1 \setminus e) = \{v_1\}$. We are assuming that v_1 only occurs in e_1 . Hence, e_1 is not $[e]$ -connected with any other edge and we, therefore, have $C_i = \{e_i\}$. In this case, \mathcal{H} is obtained by replacing H in \mathcal{H} by the hypergraphs $H_1, \dots, H_{i-1}, H_{i+1}, \dots, H_\ell$ with $E(H_j) = C_j \cup \{e\}$ for $j \neq i$. If we first apply Rule 4 to H , then we get ℓ hypergraphs H_1, \dots, H_ℓ with $E(H_i) = C_i \cup \{e_1\} = \{e, e_1\}$. We may thus delete e from H_i by Rule 2 (since we have $e \subseteq e_1$) to get H'_i and then delete H'_i altogether by Rule 5 (since we have $|E(H'_i)| = 1 \leq k$ for any $k \geq 1$). Conversely, if we first apply Rule 1 and thus delete v_1 from e_1 , then e and e_1 coincide. Hence, the resulting hypergraph only has $\ell - 1$ $[e]$ -components $C_1, \dots, C_{i-1}, C_{i+1}, \dots, C_\ell$. Rule 4 therefore allows us to replace this hypergraph by $H_1, \dots, H_{i-1}, H_{i+1}, \dots, H_\ell$ with $E(H_j) = C_j \cup \{e\}$ for $j \neq i$. Case 1.2. Suppose that $(e_1 \setminus e) \supset \{v_1\}$. Moreover, since v_1 occurs in no other edge, e_1 is connected to the other edges in C_i via vertices different from e . Hence, after deleting v_1 from e_1 , H still has ℓ $[e]$ -components $\mathcal{C}' = \{C_1, \dots, C_{i-1}, C'_i, C_{i+1}, \dots, C_\ell\}$ where $C'_i = (C_i \setminus e_1) \cup \{e'_1\}$. In this case, \mathcal{H} is obtained by replacing H in \mathcal{H} by the hypergraphs $H_1, \dots, H_{i-1}, H'_i, H_{i+1}, \dots, H_\ell$ with $E(H'_i) = C'_i \cup \{e\}$ and $E(H_j) = C_j \cup \{e\}$ for $j \neq i$. We can get these hypergraphs by first applying Rule 4 to get the hypergraphs $H_1, \dots, H_{i-1}, H_i, H_{i+1}, \dots, H_\ell$ with $E(H_i) = C_i \cup \{e\}$ and, afterwards, transforming H_i into H'_i via Rule 1. Alternatively, we can get these hypergraphs by first replacing e_1 by e'_1 in H via Rule 1 and then applying Rule 4 to get the hypergraphs $H_1, \dots, H_{i-1}, H'_i, H_{i+1}, \dots, H_\ell$ via the $[e]$ -components $\mathcal{C}' = \{C_1, \dots, C_{i-1}, C'_i, C_{i+1}, \dots, C_\ell\}$. Case 2. Now suppose $e = e_1$. Let H' with $E(H') = (E(H) \setminus \{e_1\}) \cup \{e'_1\}$. Since v_1 only occurs in e_1 , there is no difference between the $[e_1]$ -components of H and the $[e'_1]$ -components of H' . Hence, in this case, \mathcal{H} is obtained by replacing H in \mathcal{H} by the hypergraphs H'_1, \dots, H'_ℓ with $E(H'_i) = C_i \cup \{e'_1\}$ for every $i \in \{1, \dots, \ell\}$. We can get these hypergraphs by first deleting v_1 from e_1 via Rule 1 to get hypergraph H' and then applying Rule 4 to H' , where the $[e'_1]$ -components of H' are precisely $\mathcal{C} = \{C_1, \dots, C_\ell\}$. Or we may first apply Rule 4 to H to get the hypergraphs H_1, \dots, H_ℓ with $E(H_i) = C_i \cup \{e\}$ with $e = e_1$ and then apply Rule 1 to each of the resulting hypergraphs H_i and replace e_1 by e'_1 in each of them.

“(2,2)”: Suppose that two applications of Rule 2 to the same hypergraph $H \in \mathcal{H}$ are possible. That is, H contains edges e_1, e'_1 such that $e_1 \subseteq e'_1$ and edges e_2, e'_2 such that $e_2 \subseteq e'_2$. Then \mathcal{H} is obtained by replacing H in \mathcal{H} by H' such that $E(H') = E(H) \setminus \{e_1, e_2\}$. If $e'_1 \neq e_2$ and $e'_2 \neq e_1$, then it makes no difference whether we first delete e_1 or e_2 . In either case, we may afterwards delete the other edge via Rule 2.

Now suppose that $e'_1 = e_2$ holds. The case $e'_2 = e_1$ is symmetric. Then, by $e_2 \subseteq e'_2$, we also have $e_1 \subseteq e'_2$. Hence, Rule 2 is applicable to e_1, e'_2 (thus allowing us to delete e_1) and

also to e_2, e'_2 (thus allowing us to delete e_2). Hence, again, no matter whether we first delete e_1 or e_2 , we are afterwards allowed to delete also the other edge via Rule 2.

“(2,3)”: Suppose that an application of Rule 2 and an application of Rule 3 to the same hypergraph $H \in \mathcal{H}$ are possible. That is, H contains edges e_1, e_2 , such that $e_1 \subseteq e_2$ and vertices v_1, v_2 of the same type. Hence, on one hand, we may delete e_1 from H by Rule 2 and, on the other hand, we may delete v_2 from H by Rule 3.

First, suppose that $v_2 \notin e_1$. Then also $v_1 \notin e_1$. Hence, after deleting v_2 from H via Rule 3, the resulting hypergraph still contains edges e_1, e'_2 with $e_1 \subseteq e'_2$, where $e'_2 = e_2$ (if $v_2 \notin e_2$) or $e'_2 = e_2 \setminus \{v_2\}$ (if $v_2 \in e_2$). Hence, after deleting v_2 from H via Rule 3, we may still delete e_1 via Rule 2. Conversely, if we first delete e_1 from H , then v_1 and v_2 still have the same type and we may delete v_2 afterwards.

It remains to consider the case $v_2 \in e_1$. Then also $v_2 \in e_2$. Hence, after deleting v_2 from H via Rule 3, the resulting hypergraph contains the edges $e'_1 = e_1 \setminus \{v_2\}$ and $e'_2 = e_2 \setminus \{v_2\}$ with $e'_1 \subseteq e'_2$. Hence, after deleting v_2 from H via Rule 3, we may delete e'_1 via Rule 2. Conversely, if we first delete e_1 from H , then v_1 and v_2 still have the same type and we may delete v_2 afterwards.

“(2,4)”: Suppose that an application of Rule 2 and an application of Rule 4 to the same hypergraph $H \in \mathcal{H}$ are possible. That is, H contains edges e_1, e_2 , such that $e_1 \subseteq e_2$ and H contains an edge e such that H has $[e]$ -components $\mathcal{C} = \{C_1, \dots, C_\ell\}$ with $\ell \geq 2$. We distinguish several cases and subcases: Case 1. Suppose that $e_1 \neq e$. Case 1.1. If $e_1 \subseteq e$, then \mathcal{H} is obtained by replacing H in \mathcal{H} by H_1, \dots, H_ℓ with $E(H_i) = C_i \cup \{e\}$ for every $i \in \{1, \dots, \ell\}$. If we first apply Rule 4 to H , then the subedges of e are not contained in any of the components C_i . Hence, we do not even need to apply Rule 2 anymore to get rid of edge e_1 . Alternatively, if we first delete e_1 via Rule 2, then Rule 4 is still applicable to H' with $E(H') = E(H) \setminus \{e_1\}$, and we get exactly the same hypergraphs H_1, \dots, H_ℓ as before. Case 1.2. If $e_1 \not\subseteq e$, then also $e_2 \not\subseteq e$ and both e_1, e_2 are contained in exactly one $[e]$ -component C_i . In this case, \mathcal{H} is obtained by replacing H in \mathcal{H} by $H_1, \dots, H_{i-1}, H'_i, H_{i+1}, \dots, H_\ell$ with $E(H'_i) = (C_i \setminus \{e_1\}) \cup \{e\}$ and $E(H_j) = C_j \cup \{e\}$ for $j \neq i$. If we first apply Rule 4 to H , then we get the hypergraphs $H_1, \dots, H_{i-1}, H_i, H_{i+1}, \dots, H_\ell$ with $H_i = C_i \cup \{e\}$. Now Rule 2 is applicable to H_i and we may delete e_1 from H_i to get H'_i . Conversely, we may first apply Rule 2 to delete e_1 from H . Let H' with $E(H') = E(H) \setminus \{e_1\}$ denote the resulting hypergraph. Then H' has the $[e]$ -components $\mathcal{C}' = \{C_1, \dots, C_{i-1}, C'_i, C_{i+1}, \dots, C_\ell\}$ with $\ell \geq 2$ and $C'_i = C_i \setminus \{e_1\}$. Note that $C'_i \neq \emptyset$, since $e_2 \in C_i$. Hence, application of Rule 4 to H' yields the same hypergraphs $H_1, \dots, H_{i-1}, H'_i, H_{i+1}, \dots, H_\ell$ as before. Case 2. Suppose that $e_1 = e$. Then e_2 is contained in one of the $[e]$ -components. W.l.o.g., assume $e_2 \in C_\ell$. Now let $\mathcal{D} = \{D_1, \dots, D_m\}$ denote the $[e_2]$ -components of H . By $e \subseteq e_2$, every $[e_2]$ -component D_j is contained in exactly one $[e]$ -component C_i . That is, every $[e_2]$ -connected set of edges of $E(H)$ is also $[e]$ -connected but the converse is, in general, not true. Such a situation that the converse is not true may happen if a path connecting two edges uses one of the vertices in $e_2 \setminus e$. Note however that only the $[e]$ -component C_ℓ with $e_2 \in C_\ell$ contains vertices in $e_2 \setminus e$. Hence, the $[e]$ -components $C_1, \dots, C_{\ell-1}$ are also $[e_2]$ -components and we may set $D_i = C_i$ for every $i \in \{1, \dots, \ell-1\}$. For the $[e]$ -component C_ℓ , we distinguish the following 2 subcases: Case 2.1. If all edges in C_ℓ are subedges of e_2 , then the $[e_2]$ -components of H are $\mathcal{D} = \{D_1, \dots, D_{\ell-1}\}$. In this case, we obtain \mathcal{H} by replacing H in \mathcal{H} by $H_1, \dots, H_{\ell-1}$ with $E(H_i) = C_i \cup \{e\}$ for every $i \in \{1, \dots, \ell-1\}$. If we first apply Rule 4 to H , then we get the hypergraphs $H_1, \dots, H_{\ell-1}, H_\ell$ with $E(H_\ell) = C_\ell \cup \{e\}$. Since we are assuming that $e_2 \in C_\ell$

and all edges in C_ℓ are subedges of e_2 , we may apply Rule 2 to H_ℓ multiple times to delete all edges except for e_2 . Finally, when H_ℓ has been reduced to a hypergraph consisting of a single edge, we may delete H_ℓ altogether by Rule 5.

Conversely, we may first delete e from H via Rule 2. That is, we get hypergraph H' with $E(H') = E(H) \setminus \{e\}$. Then the $[e_2]$ -components of H' are simply $\mathcal{D} = \{D_1, \dots, D_{\ell-1}\}$, i.e., the subedge $e \in e_2$ is not contained in any of the $[e_2]$ -components of H anyway. Case 2.1.1. If $\ell \geq 3$, then we may apply Rule 4 to H' and replace H' by $H'_1, \dots, H'_{\ell-1}$ with $H'_i = D_i \cup \{e_2\}$. Recall that $C_i = D_i$ for every $i \in \{1, \dots, \ell - 1\}$ and that none of the vertices in $e_2 \setminus e$ occurs in C_i . Hence, each H'_i is actually of the form $H'_i = C_i \cup \{e_2\}$. Moreover, in each H'_i , the vertices in $e_2 \setminus e$ only occur in e_2 and nowhere else in H'_i . Hence, in every hypergraph H'_i , we may delete each of the vertices in $e_2 \setminus e$ via Rule 1 so that we ultimately reduce e_2 to e . That is, we transform every H'_i into H_i and we thus indeed replace H by $H_1, \dots, H_{\ell-1}$. Case 2.1.2. If $\ell = 2$, then H and also H' consists of a single $[e_2]$ -component $D_1 = C_1$. Moreover, all edges in $E(H') \setminus D_1$ are subedges of e_2 . Hence, $E(H') \setminus D_1$ is of the form $\{e_2, f_1, \dots, f_m\}$ with $m \geq 0$, such that $f_j \subseteq e_2$ holds for every j . Hence, we may delete all subedges f_j of e_2 via Rule 2 to transform H' into $D_1 \cup \{e_2\} = C_1 \cup \{e_2\}$. Then we again have the situation that all vertices in $e_2 \setminus e$ only occur in e_2 . Hence, we may delete all these vertices via multiple applications of Rule 1. In total, we may thus replace H by H_1 with $E(H_1) = C_1 \cup \{e\}$. Case 2.2. If not all edges in C_ℓ are subedges of e_2 , then C_ℓ has at least one $[e_2]$ -component. In total, the $[e_2]$ -components of H are $\mathcal{D} = \{D_1, \dots, D_{\ell-1}, D_\ell, \dots, D_m\}$ with $m \geq \ell$, such that $\{D_\ell, \dots, D_m\}$ are the $[e_2]$ -components of C_ℓ . In this case, we obtain \mathcal{H} by replacing H in \mathcal{H} by $H_1, \dots, H_{\ell-1}, H'_\ell, \dots, H'_m$ with $E(H_i) = C_i \cup \{e\}$ for every $i \in \{1, \dots, \ell - 1\}$ and $E(H'_j) = D_j \cup \{e_2\}$ for every $j \in \{\ell, \dots, m\}$. If we first apply Rule 4 (w.r.t. to edge e) to H , then we get the hypergraphs $H_1, \dots, H_{\ell-1}, H_\ell$ with $E(H_\ell) = C_\ell \cup \{e\}$. Now consider H_ℓ . Case 2.2.1. If H_ℓ consists of a single $[e_2]$ -component D_ℓ , then we simply delete all edges in $E(H_\ell) \setminus D_\ell$ to get $H'_\ell = D_\ell \cup \{e_2\}$. This is possible since all edges in $E(H_\ell) \setminus D_\ell$ are subedges of e_2 and we may therefore delete them via Rule 2. Conversely, suppose that we first delete e from H via Rule 2 to get H' with $E(H') = E(H) \setminus \{e\}$. Then we may apply Rule 4 (w.r.t. edge e_2) and replace H' by H'_1, \dots, H'_ℓ with $E(H'_i) = D_i \cup \{e_2\}$ for every $i \in \{1, \dots, \ell\}$. Again, for $i \in \{1, \dots, \ell - 1\}$, we have $D_i = C_i$ and the vertices in $e_2 \setminus e$ do not occur in C_i . Hence, in each hypergraph H'_i with $i \in \{1, \dots, \ell - 1\}$ we may delete all vertices in $e_2 \setminus e$ by multiple applications of Rule 1. In total, we thus replace H by $H_1, \dots, H_{\ell-1}, H'_\ell$ as before. Case 2.2.2. If H_ℓ consists of several $[e_2]$ -components D_ℓ, \dots, D_m with $m > \ell$, then we may apply Rule 4 to H_ℓ and replace H_ℓ by H'_ℓ, \dots, H'_m with $E(H'_j) = D_j \cup \{e_2\}$ for every $j \in \{\ell, \dots, m\}$. Conversely, suppose that we first delete e from H via Rule 2 to get H' with $E(H') = E(H) \setminus \{e\}$. Then we may apply Rule 4 (w.r.t. edge e_2) and replace H' by H'_1, \dots, H'_m with $E(H'_i) = D_i \cup \{e_2\}$ for every $i \in \{1, \dots, m\}$. Moreover, as in Case 2.2.1, every H'_i with $i \in \{1, \dots, \ell - 1\}$ can be transformed into H_i with $E(H_i) = C_i \cup \{e\}$ by deleting all vertices in $e_2 \setminus e$ via multiple applications of Rule 1 and using the equality $C_i = D_i$ for $i \in \{1, \dots, \ell - 1\}$.

“(3,3)”: Suppose that two applications of Rule 3 to the same hypergraph $H \in \mathcal{H}$ are possible. That is, H contains vertices v_1, v'_1 of the same type and vertices v_2, v'_2 of the same type. Then \mathcal{H} is obtained by replacing H in \mathcal{H} by H' such that v_1 and v_2 are deleted from all edges in H and, thus from $V(H)$ altogether. If $v'_1 \neq v_2$ and $v'_2 \neq v_1$, then it makes no difference whether we first delete v_1 or v_2 . In either case, we may afterwards also delete the other vertex via Rule 3.

Now suppose that $v'_1 = v_2$ holds. The case $v'_2 = v_1$ is symmetric. Then, all vertices v_1, v'_1, v_2, v'_2 have the same type. Hence, Rule 3 is applicable to v_1, v'_2 (thus allowing us to delete v_1) and also to v_2, v'_2 (thus allowing us to delete v_2). Hence, again, no matter whether we first delete v_1 or v_2 , we are afterwards allowed to delete also the other vertex via Rule 3.

“(3,4)”: Suppose that an application of Rule 3 and an application of Rule 4 to the same hypergraph $H \in \mathcal{H}$ are possible. That is, H contains vertices v_1, v_2 of the same type and an edge e such that H has $[e]$ -components $\mathcal{C} = \{C_1, \dots, C_\ell\}$ with $\ell \geq 2$. For any edge f , we write f' to denote $f' = f \setminus \{v_2\}$. Case 1. Suppose that $v_2 \notin e$. Then v_2 is contained in $V(C_i)$ for precisely one $[e]$ -component C_i . Moreover, since v_1 has the same type as v_2 , also the set C'_i obtained from C_i by deleting v_2 from all edges remains $[e]$ -connected. This is because that all paths that use the vertex v_2 may also use the vertex v_1 instead. Hence, after deleting v_2 from $V(H)$, H still has ℓ $[e]$ -components $\mathcal{C}' = \{C_1, \dots, C_{i-1}, C'_i, C_{i+1}, \dots, C_\ell\}$. In this case, \mathcal{H} is obtained by replacing H in \mathcal{H} by the hypergraphs $H_1, \dots, H_{i-1}, H'_i, H_{i+1}, \dots, H_\ell$ with $E(H'_i) = C'_i \cup \{e\}$ and $E(H_j) = C_j \cup \{e\}$ for $j \neq i$. We can thus get these hypergraphs by first applying Rule 4 to get the hypergraphs $H_1, \dots, H_{i-1}, H_i, H_{i+1}, \dots, H_\ell$ with $E(H_i) = C_i \cup \{e\}$ and, afterwards, transforming H_i into H'_i via Rule 3. Alternatively, we can get these hypergraphs by first deleting v_2 from H via Rule 3 and then applying Rule 4 to get the hypergraphs $H_1, \dots, H_{i-1}, H'_i, H_{i+1}, \dots, H_\ell$ via the $[e]$ -components $\mathcal{C}' = \{C_1, \dots, C_{i-1}, C'_i, C_{i+1}, \dots, C_\ell\}$. Case 2. Suppose that $v_2 \in e$. Let $e' = e \setminus \{v_2\}$. Suppose that we first transform H into H' by deleting v_2 from H via Rule 3. Then the $[e']$ -components of H' are $\mathcal{C}' = \{C'_1, \dots, C'_\ell\}$ where, for every $i \in \{1, \dots, \ell\}$, C'_i is obtained from C_i either by deleting v_2 from $V(C_i)$ if $v_2 \in V(C_i)$ or by setting $C'_i = C_i$ otherwise. Note that here we do not even make use of the fact that v_1 and v_2 have the same type. As long as a vertex $v_2 \in e$ is deleted from e and from all other edges, the $[e]$ -components of H and the $[e']$ -components of H' are exactly the same (apart from the fact, of course, that H' and, hence, its $[e']$ -components no longer contain vertex v_2). We may thus apply Rule 4 to replace H' by the set of hypergraphs $\{H'_1, \dots, H'_\ell\}$ with $H'_i = C'_i \cup \{e'\}$. Alternatively, we may first apply Rule 4 to replace H by the hypergraphs H_1, \dots, H_ℓ with $E(H_i) = C_i \cup \{e\}$. Then, in every H_i , we still have the property that v_1 and v_2 have the same type. Hence, we may apply Rule 3 to each hypergraph H_i and delete v_2 from $V(H_i)$. This results in the same set of hypergraphs $\{H'_1, \dots, H'_\ell\}$ as before.

“(4,4)”: Suppose that two applications of Rule 4 to the same hypergraph $H \in \mathcal{H}$ are possible. That is, H contains edges $e_1 \neq e_2$, such that H has $[e_1]$ -components $\mathcal{C} = \{C_1, \dots, C_\ell\}$ with $\ell \geq 2$ and H has $[e_2]$ -components $\mathcal{D} = \{D_1, \dots, D_m\}$ with $m \geq 2$. Recall that we are assuming that each hypergraph $H \in \mathcal{H}$ consists of a single connected component. Case 1. Suppose that $e_1 \subseteq e_2$ or $e_2 \subseteq e_1$ holds. The cases are symmetric, so we only need to consider $e_1 \subseteq e_2$. This case is very similar to “(2,4)”, Case 2, where e_1 now plays the role of e from “(2,4)”. Indeed, w.l.o.g., we again assume $e_2 \in C_\ell$. If Rule 4 is applied to the $[e_1]$ -components first, then we end up in precisely the same situation as with “(2,4)”. On the other hand, if Rule 4 is applied to the $[e_2]$ -components first, then all subedges of e_2 are actually deleted – including e_1 . Hence, we again end up in precisely the same situation as with “(2,4)”. Case 2. Suppose that $e_1 \not\subseteq e_2$ and $e_2 \not\subseteq e_1$ holds. Let $d = e_1 \cap e_2$. Case 2.1. Suppose that $d = \emptyset$. The edge e_1 lies in exactly one $[e_2]$ -component and e_2 lies in exactly one $[e_1]$ -component. W.l.o.g., assume $e_1 \in D_m$ and $e_2 \in C_\ell$. We claim that then all of $D_1 \cup \dots \cup D_{m-1} \cup \{e_2\}$ is contained in C_ℓ . This can be seen as follows: we are assuming that H is connected. Then also $D_1 \cup \dots \cup D_{m-1} \cup \{e_2\}$ is connected, i.e., there is a path between any two vertices in

$D_1 \cup \dots \cup D_{m-1} \cup \{e_2\}$ and this path does not need to make use of any edge in D_m . This follows immediately from the fact $V(D_m) \cap (V(D_1) \cup \dots \cup V(D_{m-1}) \cup e_2) \subseteq e_2$, which holds by the definition of components. Moreover, $e_1 \in D_m$ and we are assuming $e_1 \cap e_2 = \emptyset$. Hence, $e_1 \cap (V(D_1) \cup \dots \cup V(D_{m-1}) \cup e_2) = \emptyset$. This means that, if $D_1 \cup \dots \cup D_{m-1} \cup \{e_2\}$ is connected, then it is in fact $[e_1]$ -connected, i.e., it lies in a single $[e_1]$ -component, namely C_ℓ . By symmetry, also $C_1 \cup \dots \cup C_{\ell-1} \cup \{e_1\}$ is contained in a single $[e_2]$ -component, namely D_m .

Let \mathcal{H} be the set of hypergraphs obtained from \mathcal{H} by replacing H in \mathcal{H} by the following set of hypergraphs: $G_1, \dots, G_{\ell-1}, H_1, \dots, H_{m-1}, K$ with $E(G_i) = C_i \cup \{e_1\}$ for every $i \in \{1, \dots, \ell - 1\}$, $E(H_j) = D_j \cup \{e_2\}$ for every $j \in \{1, \dots, m - 1\}$, and $E(K) = (C_\ell \cap D_m) \cup \{e_1, e_2\}$. It remains to show that \mathcal{H} can be reached both, if Rule 4 is applied to the $[e_1]$ -components first and also if Rule 4 is applied to the $[e_2]$ -components first. Actually, \mathcal{H} is fully symmetric w.r.t. e_1 and e_2 . Hence, it suffices to show that we can reach \mathcal{H} if Rule 4 is applied to the $[e_1]$ -components of H first.

The application of Rule 4 to the $[e_1]$ -components of H allows us to replace H by G_1, \dots, G_ℓ with $E(G_i) = C_i \cup \{e_1\}$ for every $i \in \{1, \dots, \ell\}$. Next, we apply Rule 4 to the $[e_2]$ -components of G_ℓ . As was observed above, the $[e_2]$ -components D_1, \dots, D_{m-1} of H are fully contained in C_ℓ and, hence, in $E(G_\ell)$. Considering D_1, \dots, D_{m-1} as $[e_2]$ -components of G_ℓ , the application of Rule 4 gives rise to H_1, \dots, H_{m-1} with $E(H_j) = D_j \cup \{e_2\}$ for every $j \in \{1, \dots, m - 1\}$.

It remains to consider the remaining $[e_2]$ -component D_m of H , but now restricted to the hypergraph $G_\ell = C_\ell \cup \{e_1\}$. Note that it suffices to show that $(C_\ell \cap D_m) \cup \{e_1\}$ is $[e_2]$ -connected because, in this case, we would indeed get $K = (C_\ell \cap D_m) \cup \{e_1, e_2\}$ as the remaining $[e_2]$ -component when applying Rule 4 to G_ℓ . Suppose to the contrary that $(C_\ell \cap D_m) \cup \{e_1\}$ is not $[e_2]$ -connected, i.e., there exist edges $f_1, f_2 \in (C_\ell \cap D_m) \cup \{e_1\}$, such that f_1, f_2 are not $[e_2]$ -connected. We distinguish two cases: Case 2.1.1. One of the edges f_1, f_2 is e_1 , say $e_1 = f_1$. That is e_1 and f_2 are not $[e_2]$ -connected in G_ℓ . However, they are in the same $[e_2]$ -component D_m in H . This means that there is an $[e_2]$ -path in H connecting them. Since this $[e_2]$ -path is not in $C_\ell \cup \{e_1\}$, it must make use of an edge g in some $[e_1]$ -component C_i with $i \in \{1, \dots, \ell - 1\}$. W.l.o.g., assume that this path was chosen with minimal length. We can traverse this path from f_2 via g to e_1 . By assuming minimal length, the path from f_2 to g does not involve any vertex from e_1 . But then f_2 and g are $[e_1]$ -connected. This contradicts our assumption that g and f_2 lie in different $[e_1]$ -components. Case 2.1.2. Suppose that both edges f_1, f_2 are different from e_1 . Again, we have the situation that f_1 and f_2 are not $[e_2]$ -connected in G_ℓ , but they are in the same $[e_2]$ -component D_m in H . This means that there is an $[e_2]$ -path in H connecting them. Since this $[e_2]$ -path is not in $C_\ell \cup \{e_1\}$, it must make use of an edge g in some C_i with $i \in \{1, \dots, \ell - 1\}$. W.l.o.g., assume that this path was chosen with minimal length. It may possibly involve e_1 but, by the minimality, it uses e_1 at most once. If e_1 is not part of the path then we immediately get a contradiction since there exists an $[e_1]$ -path between any of the edges f_i and edge g , where f_i and g are assumed to lie in different $[e_1]$ -components. On the other hand, if e_1 is part of this path, then it must be either on the path f_1 - g or f_2 - g but not both. By symmetry, we may assume w.l.o.g., that e_1 is on the path f_1 - g . Then the path f_2 - g is an $[e_1]$ -path. Again, this contradicts our assumption that g and f_2 lie in different $[e_1]$ -components. Case 2.2. Suppose that $d \neq \emptyset$. Let R_1, \dots, R_n denote the $[d]$ -components of H . We have $d \subset e_i$ for both $i \in \{1, 2\}$, since, in Case 2, we are assuming $e_1 \not\subseteq e_2$ and $e_2 \not\subseteq e_1$. Hence, e_1 and e_2 are each contained in some $[d]$ -component. Case

2.2.1. Suppose that e_1 and e_2 are in two different $[d]$ -components. W.l.o.g., we may assume that $e_1 \in R_{n-1}$ and $e_2 \in R_n$. We observe that all $[d]$ -components except for R_{n-1} are also $[e_1]$ -components. Moreover, the remaining $[e_1]$ -components of H are entirely contained in R_{n-1} since every $[e_1]$ -component is of course also $[d]$ -connected. Let S_1, \dots, S_α with $\alpha \geq 1$ denote the $[e_1]$ -components of H inside R_{n-1} . Hence, in total, H has the $[e_1]$ -components $R_1, \dots, R_{n-2}, R_n, S_1, \dots, S_\alpha$.

Likewise, all $[d]$ -components except for R_n are also $[e_2]$ -components and the remaining $[e_2]$ -components of H are entirely contained in R_n . Let T_1, \dots, T_β with $\beta \geq 1$ denote the $[e_2]$ -components of H inside R_n . Hence, in total, H has the $[e_2]$ -components $R_1, \dots, R_{n-1}, T_1, \dots, T_\beta$.

Let \mathcal{H} be the set of hypergraphs obtained from \mathcal{H} by replacing H in \mathcal{H} by the following set of hypergraphs: $F_1, \dots, F_{n-2}, G_1, \dots, G_\alpha, H_1, \dots, H_\beta$ with $F_i = R_i \cup \{d\}$ for every $i \in \{1, \dots, n-2\}$, $G_i = S_i \cup \{e_1\}$ for every $i \in \{1, \dots, \alpha\}$, $H_i = T_i \cup \{e_2\}$ for every $i \in \{1, \dots, \beta\}$. It remains to show that \mathcal{H} can be reached both, if Rule 4 is applied to the $[e_1]$ -components first and also if Rule 4 is applied to the $[e_2]$ -components first. Actually, \mathcal{H} is fully symmetric w.r.t. e_1 and e_2 . Hence, it suffices to show that we can reach \mathcal{H} if Rule 4 is applied to the $[e_1]$ -components of H first.

As observed above, the $[e_1]$ -components of H are $R_1, \dots, R_{n-2}, R_n, S_1, \dots, S_\alpha$. Hence, we may replace H by the hypergraphs $F'_1, \dots, F'_{n-2}, F'_n, G_1, \dots, G_\alpha$, where the G_i 's are defined as above and the hypergraphs F'_i with $i \neq n-1$ are obtained as $E(F'_i) = R_i \cup \{e_1\}$. By assumption, e_1 is in the $[d]$ -component R_{n-1} . Hence, $e_1 \cap V(R_i) \subseteq d$ for all $i \neq n-1$. In other words, the vertices in $e_1 \setminus d$ only occur in a single edge of F'_i with $i \neq n-1$, namely in the edge e_1 . We may therefore apply Rule 1 multiple times to each of the hypergraphs F'_i with $i \neq n-1$. In this way, we replace e_1 in each of these hypergraphs by d and we indeed transform F'_i into F_i for every $i \leq n-2$.

Also in $F'_n = R_n \cup \{e_1\}$ we thus replace e_1 by d . Recall that we are assuming that $e_2 \in R_n$. Hence, we may delete d by Rule 2 since, $d \subseteq e_2$. Hence, F'_n is ultimately transformed into R_n . Now consider the $[e_2]$ -components of H inside R_n , namely T_1, \dots, T_β with $\beta \geq 1$. These are also the $[e_2]$ -components of the hypergraph R_n , i.e., $T_i \subseteq E(R_n)$ and T_i is (maximally) $[e_2]$ -connected for every i . If $\beta \geq 2$, then we may apply Rule 4 to R_n and we get precisely the desired hypergraphs $H_i = T_i \cup \{e_2\}$ for every $i \in \{1, \dots, \beta\}$. On the other hand, if $\beta = 1$, then R_2 has a single $[e_2]$ -component T_1 . Note that all edges of a hypergraph not contained in any of its $[e_2]$ -components are subedges of e_2 . Hence, we may again transform R_n into $H_1 = T_1 \cup \{e_2\}$ by multiple applications of Rule 2, which allows us to delete all subedges of e_2 . Case 2.2.2. Suppose that e_1 and e_2 are in the same $[d]$ -component. W.l.o.g., we may assume that $\{e_1, e_2\} \subseteq R_n$. We observe that all $[d]$ -components except for R_n are also $[e_1]$ -components and $[e_2]$ -components. Moreover, the remaining $[e_1]$ -components of H and also the remaining $[e_2]$ -components of H are entirely contained in R_n . Let S_1, \dots, S_α with $\alpha \geq 1$ denote the $[e_1]$ -components of H inside R_n and let T_1, \dots, T_β with $\beta \geq 1$ denote the $[e_2]$ -components of H inside R_n . Then, in total, H has the $[e_1]$ -components $R_1, \dots, R_{n-1}, S_1, \dots, S_\alpha$ and the $[e_2]$ -components $R_1, \dots, R_{n-1}, T_1, \dots, T_\beta$.

We are assuming that $\{e_1, e_2\} \subseteq R_n$. Hence, e_1 is in precisely one $[e_2]$ -component T_j inside R_n and e_2 is in precisely one $[e_1]$ -component S_i inside R_n . W.l.o.g., we may assume that $e_1 \in T_\beta$ and $e_2 \in S_\alpha$. Analogously to the Case 2.1, we claim that then all of $T_1 \cup \dots \cup T_{\beta-1} \cup \{e_2\}$ is contained in S_α . This can be seen as follows: we are assuming that

H is connected. Then also $T_1 \cup \dots \cup T_{\beta-1} \cup \{e_2\}$ is connected and even $[e_1]$ -connected, since $e_1 \in T_\beta$. Hence, $T_1 \cup \dots \cup T_{\beta-1} \cup \{e_2\}$ lies in a single $[e_1]$ -component, namely S_α . By symmetry, also $S_1 \cup \dots \cup S_{\alpha-1} \cup \{e_1\}$ is contained in a single $[e_2]$ -component, namely T_β .

We now define the set \mathcal{H} of hypergraphs by combining the ideas of the Cases 2.1 and 2.2.1. Let \mathcal{H} be the set of hypergraphs obtained from \mathcal{H} by replacing H in \mathcal{H} by the following set of hypergraphs: $F_1, \dots, F_{n-1}, G_1, \dots, G_{\alpha-1}, H_1, \dots, H_{\beta-1}, K$ with $E(F_i) = R_i \cup \{d\}$ for every $i \in \{1, \dots, n-1\}$, $E(G_i) = S_i \cup \{e_1\}$ for every $i \in \{1, \dots, \alpha-1\}$, $E(H_i) = T_i \cup \{e_2\}$ for every $i \in \{1, \dots, \beta-1\}$, and $E(K) = (S_\alpha \cap T_\beta) \cup \{e_1, e_2\}$. It remains to show that \mathcal{H} can be reached both, if Rule 4 is applied to the $[e_1]$ -components first and also if Rule 4 is applied to the $[e_2]$ -components first. Again, since \mathcal{H} is fully symmetric w.r.t. e_1 and e_2 , it suffices to show that we can reach \mathcal{H} if Rule 4 is applied to the $[e_1]$ -components of H first.

As observed above, the $[e_1]$ -components of H are $R_1, \dots, R_{n-1}, S_1, \dots, S_\alpha$. Hence, we may replace H in \mathcal{H} by the hypergraphs $F'_1, \dots, F'_{n-1}, G_1, \dots, G_\alpha$, where the hypergraphs F'_i with $i \leq n-1$ are obtained as $E(F'_i) = R_i \cup \{e_1\}$ and $E(G_\alpha) = S_\alpha \cup \{e_1\}$. For $i \in \{1, \dots, \alpha-1\}$, G_i is as defined above, i.e., $E(G_i) = S_i \cup \{e_1\}$. By assumption, e_1 is in the $[d]$ -component R_n . Hence, $e_1 \cap V(R_i) \subseteq d$ for all $i \leq n-1$. In other words, the vertices in $e_1 \setminus d$ only occur in a single edge of F'_i with $i \leq n-1$, namely in the edge e_1 . We may therefore apply Rule 1 multiple times to each of the hypergraphs F'_i with $i \leq n-1$. In this way, we replace e_1 in each of these hypergraphs by d and we indeed transform F'_i into F_i for every $i \leq n-1$.

Now consider the hypergraph G_α with $E(G_\alpha) = S_\alpha \cup \{e_1\}$. We apply Rule 4 to the $[e_2]$ -components of G_α . As was observed above, the $[e_2]$ -components $T_1, \dots, T_{\beta-1}$ of H are fully contained in S_α and, hence, in $E(G_\alpha)$. Considering $T_1, \dots, T_{\beta-1}$ as $[e_2]$ -components of G_α , the application of Rule 4 gives rise to $H_1, \dots, H_{\beta-1}$ with $E(H_i) = T_i \cup \{e_2\}$ for every $i \in \{1, \dots, \beta-1\}$.

It remains to consider the remaining $[e_2]$ -component T_β of H , but now restricted to the hypergraph $G_\alpha = S_\alpha \cup \{e_1\}$. It suffices to show that $(S_\alpha \cap T_\beta) \cup \{e_1\}$ is $[e_2]$ -connected because, in this case, we would indeed get $(S_\alpha \cap T_\beta) \cup \{e_1\}$ as the remaining $[e_2]$ -component when applying Rule 4 to G_α , and K with $E(K) = (S_\alpha \cap T_\beta) \cup \{e_1, e_2\}$ would be the remaining hypergraph produced by this application of Rule 4. The proof follows the same line of argumentation as Case 2.1. More specifically, assume to the contrary that there are two edges f_1, f_2 in $(S_\alpha \cap T_\beta) \cup \{e_1\}$, such that f_1, f_2 are not $[e_2]$ -connected in G_α . However, f_1, f_2 are in the same $[e_2]$ -component T_β of H . Hence, there exists a path between f_1 and f_2 using an edge from some $[e_1]$ -component different from S_α . This can be exploited to derive a contradiction by constructing an $[e_1]$ -path between two different $[e_1]$ -components. For details, see Case 2.1.

4 Parallelisation strategy

As described in more detail below, we use a divide and conquer method, based on the balanced separator approach. This method divides a hypergraph into smaller hypergraphs, called *subcomponents*. Our method proceeds to work on these subcomponents in parallel, with each round reducing the size of the hypergraphs (i.e., the number of edges in each subcomponent) to at most half their size. Thus after logarithmically many rounds, the method will have decomposed the entire hypergraph, if

a decomposition of width k exists. For the computation we use the modern programming language Go [10], which has a model of concurrency based on [28].

In Go, a *goroutine* is a sequential process. Multiple goroutines may run concurrently. In the pseudocode provided, these are spawned using the keyword **go**, as can be seen in Algorithm 1, line 16. They communicate over *channels*. Using a channel ch is indicated by $\leftarrow ch$ for *receiving* from a channel, and by $ch \leftarrow$ for *sending* to ch .

4.1 Overview

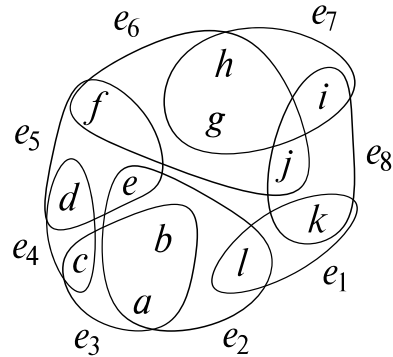
Algorithm 1 contains the full decomposition procedure, whereas Function FindBalSep details the parallel search for separators, as it is a key subtask for parallelisation. To emphasise the core ideas of our parallel algorithm, we present it as a decision procedure, which takes as input a hypergraph H and a parameter k , and returns as output either *Accept* if $ghw(H) \leq k$ or *Reject* otherwise. Please note, however, that our actual implementation also produces a GHD of width $\leq k$ in case of an accepting run.

For the GHD computation, we may assume w.l.o.g. that the input hypergraph has no isolated vertices (i.e., vertices that do not occur in any edge). Hence, we may identify H with its set of edges $E(H)$ with the understanding that $V(H) = \bigcup E(H)$ holds. Likewise, we may consider a subhypergraph H' of H as a subset $H' \subseteq H$ where, strictly speaking, $E(H') \subseteq E(H)$ holds.

Our parallel Balanced Separator algorithm begins with an initial call to the procedure `Decomp`, as seen in line 2 of Algorithm 1. The procedure `Decomp` takes two arguments, a subhypergraph H' of H for the current subcomponent considered, and a set Sp of “special edges”. These special edges indicate the balanced separators encountered so far, as can be seen in line 16, where the current separator *subSep* is added to the argument on the recursive call, combining all its vertices into a new special edge. The special edges are needed to ensure that the decompositions of subcomponents can be combined to an overall decomposition, and are thus considered as additional edges. The goal of procedure `Decomp` is to find a GHD \mathcal{D} of $H' \cup Sp$ in such a way that each special edge $s \in Sp$ must be “covered” by a leaf node n_s of \mathcal{D} with the properties $\lambda(n_s) = \{s\}$ and $\chi(n_s) = s$ and s may not occur in the λ -label of any other node in the GHD, i.e., we may only use edges from H for these λ -labels. Thus the set Sp imposes additional conditions on the GHD. In the sequel, we shall refer to a pair (H', Sp) consisting of a subhypergraph H' of H and a set of special edges Sp as an “extended subhypergraph” of H . Clearly, also H itself together with the empty set of special edges is an extended subhypergraph of itself and a GHD of H also satisfies the additional conditions of a GHD of the extended subhypergraph (H, \emptyset) . Hence, the central procedure `Decomp` in Algorithm 1, when initially called on line 2, checks if there exists a GHD of desired width of the extended subhypergraph (H, \emptyset) , that is, a GHD of hypergraph H itself.

The recursive procedure `Decomp` has its base case in lines 4 to 5, when the size of H' and Sp together is less than or equal to 2. The remainder of `Decomp` consists of two loops, the Separator Loop, from lines 7 to 24, which iterates over all balanced separators, and within it the SubEdge Loop, running from lines 12 to 23, which

Fig. 3 An example hypergraph, where the vertices are represented by letters, with explicit edge names



iterates over all subedge variants of any balanced separator. New balanced separators are produced with the subprocedure `FindBalSep`, used in line 8 of Algorithm 1, and detailed in Function `FindBalSep`. After a separator is found, `Decomp` computes the new subcomponents in line 13. Then goroutines are started using recursive calls of `Decomp` for all found subcomponents. If any of these calls returns `Reject`, seen in line 19, then the procedure starts checking for subedges. If they have been exhausted, the procedure checks for another separator. If all balanced separators have been tried without success, then `Decomp` rejects in line 25.

We proceed to detail the parallelisation strategy of the two key subtasks: the search for new separators and the recursive calls over the subcomponents created from a chosen separator.

4.2 Parallel search for balanced separators

Before describing our implementation, we define some needed notions. For the search for balanced separators within an extended subhypergraph $H' \cup Sp$, we can determine the set of relevant edges from the hypergraph, defined as $E^* = \{e \in E(H) \mid e \cap \bigcup(H' \cup Sp) \neq \emptyset\}$. We assume for this purpose that the edges in E^* are ordered and carry indices in $\{1, \dots, |E^*|\}$ according to this ordering. We can then define the following notion.

Definition 1 A *k-combination* for an ordered set of edges E^* is a k -tuple of integers (x_1, \dots, x_k) , where $1 \leq x_i \leq |E^*|$ and $x_1 < \dots < x_k$. For two k -combinations a, b , we say b is one step ahead of a , denoted as $a <_1 b$, if w.r.t. the lexicographical ordering $<_{lex}$ on the tuples, we have $a <_{lex} b$, and there exists no other k -combination c s.t. $a <_{lex} c <_{lex} b$. To generalise, we say c is i steps ahead of a with $i > 1$, if there exists some b s.t. $a <_{i-1} b <_1 c$.

Example 3 Consider the hypergraph H from Fig. 3. Assume that we are currently investigating the extended subhypergraph with $H' = \{e_3, e_4, e_5\}$ and $Sp = \{\{a, b, e, f\}\}$. By the

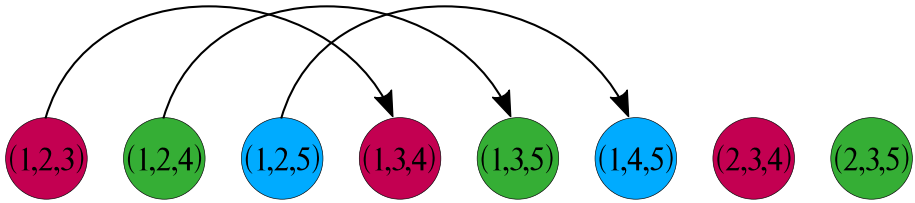


Fig. 4 Using k -combinations to split the workspace. Shown here with 3 workers, and $k = 3$ and $|E^*| = 5$

definition above, this gives us the following set of relevant edges: $E^* = \{e_2, e_3, e_4, e_5, e_6\}$. We assume the ordering to be simply the order the edges are written in here, i.e., index 1 refers to edge e_2 , 2 refers to e_3 , etc.

Let us assume that we are looking for separators of length 3, so $k = 3$. We would then start the search with the 3-combination (1,2,3), which represents the choice of e_2, e_3, e_4 . If we move one step ahead, we next get the 3-combination (1,2,4), which represents the choice of e_2, e_3, e_5 . Moving further 3 steps ahead, we produce the 3-combination (1,3,5), representing the choice of e_1, e_4, e_6 .

In our parallel implementation, while testing out a number of configurations, we settled ultimately on the following scenario, shown in Function FindBalSep: we first create w many worker goroutines, seen in lines 3 to 4, where w stands for the number of CPUs we have available. This corresponds to splitting the workspace into w parts, and assigning each of them to one worker. Each worker is passed two arguments:

First, the workers are passed a channel ch , which they will use to send back any balanced separators they find. The worker procedure iterates over all candidate separators in its assigned search space, and sends back the first balanced separator it finds over the channel.

Secondly, to coordinate the search, each worker is passed a k -combination, where the needed ordering is on the relevant edges defined earlier. Furthermore, each worker starts with a distinct offset of j steps ahead, where $0 \leq j \leq w - 1$, and will only check k -combinations that are w steps apart each. This ensures that no worker will redo the work of another one, and that together they still cover the entire search space. An illustration for this can be seen in Fig. 4.

Having started the workers, FindBalSep then waits for one of two conditions (whichever happens first): either one of the workers finds a balanced separator, lines 6 to 7, or none of them does and they all terminate on their own once they have exhausted the search space. Then the empty set is returned, as seen in line 8, indicating that no further balanced separators from edges in E^* exist. We note that balanced separators composed from subedges are taken care of in Algorithm 1 on lines 19 to 21, and are therefore not relevant for the search inside the Function FindBalSep.

We proceed to explain how this design addresses the three challenges for a parallel implementation we outlined in the introduction.

- i This design reduces the need for synchronisation: each worker is responsible for a share of the search space, and the only time a worker is stopped is when either it has found a balanced separator, or when another worker has done so.
- ii The number of worker goroutines scales with the number of available processors. This allows us to make use of the available hardware when searching for balanced separators,

and the design above makes it very easy to support an arbitrary number of processors for this, without a big increase in the synchronisation overhead.

- iii Finally, our design addresses backtracking in this way: as explained, the workers employ a set of k -combinations, called M in Function `FindBalSep`, to store their current progress, allowing them to generate the next separator to consider. Crucially, this data structure is stored in `Decomp`, seen in line 6 of Algorithm 1, *even after the search is over*. Therefore, in case we need to backtrack, this allows the algorithm to quickly continue the search exactly where it left off, without losing any work. If multiple workers find a balanced separator, one of them arbitrarily “wins”, and during backtracking, the other workers can immediately send their found separators to `FindBalSep` again.

Algorithm 1 Parallel balanced separator algorithm.

```

Type:  $\text{Comp} = (H: \text{Graph}, Sp: \text{Set of Special Edges})$ 
Input:  $H$ : Hypergraph
Parameter:  $k$ : width parameter
Output: Accept if  $ghw$  of  $H \leq k$ , else Reject

1 begin
2   return Decomp ( $H, \emptyset$ ) ▷ initial call
3 Function Decomp ( $H': \text{Graph}, Sp: \text{Set of Special Edges}$ )
4   if  $|H' \cup Sp| \leq 2$  then ▷ Base Case
5     return Accept
6    $M :=$  a set of  $k$ -tuples ▷ used to facilitate fast backtracking
7   repeat ▷ SeparatorLoop
8      $sep := \text{FindBalSep}(H', Sp, M)$ 
9     if  $sep = \emptyset$  then
10       break
11      $subSep := sep$ 
12     repeat ▷ SupEdgeLoop
13        $comps := \text{ComputeSubhypergraphs}(H', Sp, subSep)$  ▷ returns
14        $Comp$  set
15        $ch :=$  a channel
16       for  $c \in comps$  do
17         go  $ch \leftarrow \text{Decomp}(c.H, c.Sp \cup \bigcup subSep)$ 
18       while any recursive call is still running do
19          $out \leftarrow ch$ ; ▷ waits on channel
20         if  $out = \text{Reject}$  then
21            $subSep = \text{NextSubedgeSep}(subSep)$ 
22           continue SubEdgeLoop
23         return Accept ▷ found decomposition
24     until  $subSep = \emptyset$ 
25   until  $sep = \emptyset$ 
26   return Reject ▷ exhausted search space

```

Function FindBalSep(H' , Sp, M).

```

1 Function FindBalSep( $H'$ : Graph, Sp: Set of Special Edges, M: Set of  $k$ -tuples)
2    $ch :=$  a channel
3   for  $M_i \in M$  do
4     go WORKER( $H'$ , Sp,  $M_i$ ,  $ch$ )
5   while any worker still running do
6     out  $\leftarrow ch$ : ▷ wait on channel
7     return out
8   return empty set ▷ exhausted search space
9 Function Worker( $H'$ : Graph, Sp: Set of Special Edges,  $M_i$ :  $k$ -tuple of integers,  $ch$ :
   Channel)
10  for  $sep \in \text{NextSeparator}(M_i)$  do
11    if IsBalanced( $sep$ ,  $H'$ , Sp) then
12       $ch \leftarrow sep$  ▷ send  $sep$  to FindBalSep
13  return ▷ no separator found within  $M_i$ 

```

4.3 Parallel recursive calls

For the recursive calls on the produced subcomponents, we create for each such call its own goroutine, as explained in the overview. This can be seen in Algorithm 1, line 15, where the output is then sent back via the channel ch . Each call gets as arguments its own extended subhypergraph, as well as an additional special edge. The output is received at line 18, where the algorithm waits on all recursive calls to finish before it can either return accept, or reject the current separator in case any recursive call returns a reject.

The fact that all recursive calls can be worked on concurrently is also in itself a major performance boost: in the sequential case we execute all recursive calls in a loop, but in the parallel algorithm - see lines 14 to 15 in Algorithm 1 - we can execute these calls simultaneously. Thus, if one parallel call rejects, we can stop all the other calls early, and thus potentially save a lot of time. It is easy to imagine a case where in the sequential execution, a rejecting call is encountered only near the end of the loop.

We state how we addressed the challenges of parallelisation in this area:

- i Making use of goroutines and channels makes it easy to avoid any interference between the recursive calls, and the design allows each recursive call to run and return its results fully independently. Thus when running the recursive calls concurrently, we do not have to make use of synchronisation at all.
- ii The second challenge, scaling with the number of CPUs, is initially limited by the number of recursive calls, which itself is dependent on the number of connected components. We can ensure, however, that we will generally have at least two, unless we manage to

cover half the graph with just k edges. While this might look problematic at first, each of these recursive calls will either hit a base case, or once more start a search for a balanced separator which as outlined earlier, will always be able to make use of all cores in our CPU. This construction is aided by the fact that Go can easily manage a very large number of goroutines, scheduling them to make optimal use of the available resources. Thus our second challenge has also been addressed.

- iii The third challenge, regarding backtracking, was written with the search for a balanced separator in mind, and is thus not directly applicable to the calls of the procedure `Decomp`. To speed up backtracking also in this case, we did initially consider the use of caching – which was used to great effect in `det- k -decomp` [23]. The algorithm presented here, however, differs significantly from `det- k -decomp` by the introduction of special edges. This makes cache hits very unlikely, since both the subhypergraph H' and the set of special edges Sp must coincide between two calls of `Decomp`, to reuse a previously computed result from the cache. Hence, caching turned out to be not effective here.

Another important topic concerns the scheduling of goroutines. This is relevant for us, since during every recursive call, we start as many goroutines as there are CPUs. Luckily, Go implements a so-called “work stealing” scheduler, which allows idle CPUs to take over parts of the work of other CPUs. Since goroutines have less of an overhead than normal threads, we can be sure that our algorithm maximally utilises the given CPU resources, without creating too much of an overhead. For more information about the scheduling of goroutines, we refer to the handbook by Cox-Buday [9].

To summarise, two of the challenges were addressed and solved, while the third, which mainly targeted the search for a balanced separator, was not applicable here. The parallelisation of recursive calls therefore gives a decent speed-up as will be illustrated by the experimental results in Section 5.

4.4 Correctness of the parallel algorithm

It is important to note that this parallel algorithm is a correct decomposition procedure. More formally, we state the following property:

Theorem 3 *The algorithm for checking the ghw of a hypergraph given in Algorithm 1 is sound and complete. More specifically, Algorithm 1 with input H and parameter k will accept if and only if there exists a GHD of H with width $\leq k$. Moreover, by materialising the decompositions implicitly constructed in the recursive calls of the `Decomp` function, a GHD of width $\leq k$ can be constructed efficiently in case the algorithm returns `Accept`.*

Proof A sequential algorithm for GHD computation based on balanced separators was first presented in [14]. Let us refer to it as `SequentialBalSep`. A detailed proof of the soundness and completeness of `SequentialBalSep` is given in [14]. For convenience of the reader, we recall the pseudo-code description of `SequentialBalSep` from [14] in Appendix A. In order to prove the soundness and completeness of our parallel algorithm for GHD computation, it thus suffices to show that, for every hypergraph H and integer $k \geq 1$, our algorithm returns `Accept` if and only if `SequentialBalSep` returns a GHD of H of width $\leq k$. Hence, since both algorithms operate on the same notion of extended

subhypergraphs and their GHDs, we have to show that, for every $k \geq 1$ and every input (H', Sp) , the `Decomp` function of our algorithm returns `Accept` if and only if the `Decompose` function of the `SequentialBalSep` algorithm returns a GHD of $H' \cup Sp$ of width $\leq k$.

To prove this equivalence between our new parallel algorithm and the previous `SequentialBalSep` algorithm from [14], we inspect the main differences between the two algorithm and argue that they do not affect the equivalence:

1. *Decision problem vs. search problem.* While `SequentialBalSep` outputs a concrete GHD of desired width if it exists, we have presented our algorithm as a pure decision procedure which outputs `Accept` or `Reject`. Note that this was only done to simplify the notation. It is easy to verify that the construction of a GHD in the `SequentialBalSep` algorithm on lines 5 – 12 (for the base case) and on line 27 (for the inductive case) can be taken over literally for our parallel algorithm.
2. *Parallelisation.* The most important difference between the previous sequential algorithm and the new parallel algorithm is the parallelisation. As was mentioned before, parallelisation is applied on two levels: splitting the search for finding the next balanced separator into parallel subtasks via function `FindBalSep` and processing recursive calls of function `Decomp` in parallel. The parallelisation via function `FindBalSep` will be discussed separately below. We concentrate on the recursive calls of function `Decomp` first. On lines 13 – 22 of our parallel algorithm, function `Decomp` is called recursively for all components of a given balanced separator and `Accept` is returned on line 22 if and only if all these recursive calls are successful. Otherwise, the next balanced separator is searched for. The analogous work is carried out on lines 18 – 27 of the `SequentialBalSep` algorithm. That is, the function `Decompose` is called recursively for all components of a given balanced separator and (by combining the GHDs returned from these recursive calls) a GHD of the given extended subhypergraph is returned on line 27 if and only if all these recursive calls are successful. Otherwise, the next balanced separator is searched for.
3. *Search for balanced separators.* As has been detailed in Section 4.2, our function `FindBalSep` splits the search space for a balanced separator into w pieces (where w denotes the number of available workers) and searches for a balanced separator in parallel. So in principle, this function has the same functionality as the iterator `BalSepIt` in the `SequentialBalSep` algorithm. That is, the set of balanced separators of size k for an extended subhypergraph $H' \cup Sp$ found is the same when one calls the function `FindBalSep` until it returns the empty set, or when one calls the iterator `BalSepIt` until it has no elements to return any more. However, the calls of function `FindBalSep` implement one of the algorithmic improvements presented in Section 3.2: note that the `SequentialBalSep` algorithm assumes that all required subedges of edges from $E(H)$ have been added to $E(H)$ before executing this algorithm. It may thus happen that, by considering different subedges of a given k -tuple of edges, the same separator (i.e., the same set of vertices) is obtained several times. As has been explained in Section 3.2, we avoid this repetition of work by concentrating on the set of vertices of a given balanced separator (i.e., `sep` returned on line 8 and used to initialize `subSep` on line 11)

and iterate through all balanced separators obtained as “legal” subsets by calling the `NextSubedgeSep` function on line 20. This means that we ultimately get precisely the same balanced separators (considered as sets of vertices) as in the `SequentialBalSep` algorithm.

4.5 Hybrid approach - best of both worlds

Based on this parallelisation scheme, we produced a parallel implementation of the Balanced Separator algorithm, with the improvements mentioned in Section 3. We already saw some promising results, but we noticed that for many instances, this purely parallel approach was not fast enough. We thus continued to explore a more nuanced approach, mixing both parallel and sequential algorithms.

We now present a novel combination of parallel and sequential decomposition algorithms. It contains all the improvements mentioned in Section 3 and combines the Balanced Separator algorithm from Sections 4.1–4.3 and *det-k-decomp* recalled in Section 2.1.

This combination is motivated by two observations: The Balanced Separator algorithm is very effective at splitting large hypergraphs into smaller ones and in negative cases, where it can quickly stop the computation if no balanced separator for a given subcomponent exists. It is slower for smaller instances where the computational overhead to find balanced separators at every step slows things down. Furthermore, for technical reasons, it is also less effective at making use of caching. *det-k-decomp*, on the other hand, with proper heuristics, is very efficient for small instances and it allows for effective caching, thus avoiding repetition of work.

The Hybrid approach proceeds as follows: For a fixed number m of rounds, the algorithm tries to find balanced separators. Each such round is guaranteed to halve the number of hyperedges considered. Hence, after those m rounds, the number of hyperedges in the remaining subcomponents will be reduced to at most $\frac{|E(H)|}{2^m}$. The Hybrid algorithm then proceeds to finish the remaining subcomponents by using the *det-k-decomp* algorithm.

This required quite extensive changes to *det-k-decomp*, since it must be able to deal with Special Edges. Formally, each call of *det-k-decomp* runs sequentially. However, since the m rounds can produce a number of components, many calls of *det-k-decomp* can actually run in parallel. In other words, our Hybrid approach also brings a certain level of parallelism to *det-k-decomp*.

5 Experimental evaluation and results

We have performed our experiments on the HyperBench benchmark from [14] with the goal to determine the exact generalized hypertree width of significantly more instances. We thus evaluated how our approach compares with existing attempts to compute the *ghw*, and we investigated how various heuristics and parameters prove beneficial for various instances. The detailed results of our experiments¹, in addition to the source code of our Go programs² are publicly available. Together with the benchmark instances, which

¹ raw data available under [22]

² See: <https://github.com/cem-okumus/BalancedGo>

Table 2 Overview of the instances from HyberBench and their average sizes by group, as well as sizes of groups themselves

Group	Instances				
	Avg sizes		Arity		Size
	$ V $	$ E $	avg	max	
CSP Application	151.71	68.90	7.00	35	1090
CSP Random	40.74	67.58	4.85	15	863
CSP Other	372.40	395.68	4.24	14	82
CQ Application	30.88	7.03	11.03	145	1113
CQ Random	47.99	27.54	10.63	20	500
Total	79.34	51.39	8.15	145	3648

are detailed below and also publicly available, this ensures the reproducibility of our experiments.

5.1 Benchmark instances and setting

HyperBench The instances used in our experiments are taken from the benchmark HyperBench, collected from various sources in industry and the literature, which was released in [14] and made publicly available at <http://hyperbench.dbai.tuwien.ac.at>. It consists of 3648 hypergraphs from CQs and CSPs, where for many CSP instances the exact *ghw* was still undetermined. In this extended evaluation, we performed the evaluation on a larger set of instances when compared with the original paper [21], to reflect the newest version of the benchmark, published in [14]. We provide a more detailed overview of the various instances, grouped by their origin, in Table 2. The first two columns of “Avg sizes”, refer to the sizes of instances within the groups, and the final column “Size” refers to the cardinality of the group, i.e. how many instances it includes. The two “Arity” columns refer to the maximum and average edge sizes of the hypergraphs in each group.

Hardware and software We used Go 1.2 for our implementation, which we refer to as *BalancedGo*. Our experiments ran on a cluster of 12 nodes, running Ubuntu 16.04.1 LTS with a 24 core Intel Xeon E5-2650v4 CPU, clocked at 2.20 GHz, each node with 256 GB of RAM. We disabled HyperThreading for the experiments.

Setup and limits For the experiments, we set a number of limits to test the efficiency of our solution. For each run, consisting of the input (i.e., hypergraph H and integer k) and a configuration of the decomposer, we set a one hour (3600 seconds) timeout and limited the available RAM to 1 GB. These limits are justified by the fact that these are the same limits as were used in [14], thus ensuring the direct comparability of our test results. To enforce these limits and run the experiments, we used the *HTCondor* software [39], originally named just Condor. Note that for the test results of HtdSMT, we set the available RAM to 24 GB, as that particular solver had a much higher memory consumption during our tests.

Table 3 Overview of previous results: number of instances solved and running times (in seconds) for producing optimal-width GHDs in [14] and [38]

Instances	Decomposition Methods							
Group	NewDetKDecomp by [14]				HtdSMT by [38]			
	#solved	avg	max	stdev	#solved	avg	max	stdev
CSP Application	386	150.82	2608.0	490.47	571	227.27	3508.5	529.90
CSP Random	412	65.78	3240.0	379.12	587	366.93	3569.0	756.10
CSP Other	27	126.43	2538.0	422.42	23	371.77	3340.3	728.27
CQ Application	1113	0.00	0.0	0.00	1070	32.00	1437.0	113.60
CQ Random	281	2.12	335.0	21.14	254	192.30	3486.5	552.20
Total	2219	59.00	3240.0	325.03	2505	158.25	3569.0	481.64

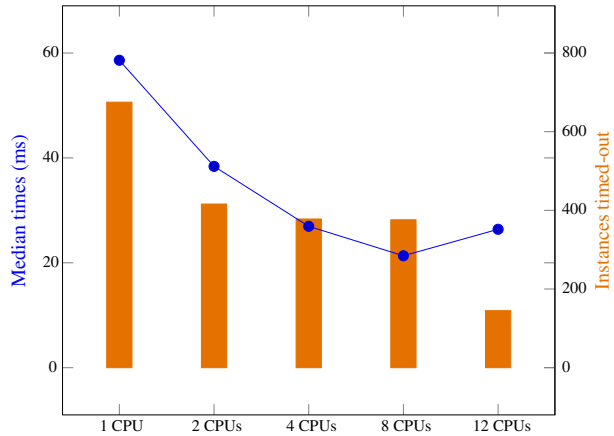
Table 4 Overview of our results: number of instances solved and running times (in seconds) for producing optimal-width GHDs by our new algorithms

Instances	Decomposition Methods							
Group	Hybrid Approach				BalancedGo <i>ensemble</i>			
	#solved	avg	max	stdev	#solved	avg	max	stdev
CSP Application	762	6.24	3247.90	80.70	763	30.86	3572.78	211.83
CSP Random	578	29.31	3589.82	246.19	625	48.60	3589.82	297.21
CSP Other	42	34.33	2236.00	194.52	42	45.86	2438.64	223.75
CQ Application	1113	0.00	0.01	0.00	1113	0.00	1.74	0.02
CQ Random	355	16.45	3574.76	198.97	381	27.87	3574.76	231.01
Total	2850	11.30	3589.82	145.47	2924	25.76	3589.82	207.32

5.2 Empirical results

The key results from our experiments are summarised in Table 4, with Table 3 acting as a comparison point. Under “Decomposition Methods” we use “*ensemble*” to indicate that results from multiple algorithms are collected, i.e., results from the Hybrid algorithm, the parallel Balanced Separator algorithm and det-*k*-decomp. To also consider the performance of one of the individual approaches introduced in Section 4, namely the results of the Hybrid approach (from Section 4.5) is separately shown in a section of the table. As a reference point, we considered on one hand the *NewDetKDecomp* library from [14] and also the SAT Modulo Theory based solver *HtdSMT* from [38]. For each of these, we also listed the average time and the maximal time to compute a GHD of optimal-width for each group of instances of HyperBench, as well as the standard deviation. The minimal times are left out for brevity, since they are always near or equal to 0. Note that for HyperBench the instance groups “CSP Application” or “CQ Application”, listed in Tables 3 and 4 are hypergraphs of (resp.) CSP or CQ instances from real world applications.

Fig. 5 Study of the performance gain w.r.t. the number of CPUs used



In the left part of Table 4, we report on the following results obtained with our Hybrid Approach described in Section 4.5, while the right part of that table shows the result for the “BalancedGo ensemble”. Recall that by “ensemble” we mean the combination of the information gained from runs of all our decomposition algorithms. For a hypergraph H and a width k , an accepting run gives us an upper bound (since the optimal $ghw(H)$ is then clearly $\leq k$), and a rejecting run gives us a lower bound (since then we know that $ghw(H) > k$). By pooling multiple algorithms, we can combine these produced upper and lower bounds to compute the optimal width (when both bounds meet) for more instances than any one algorithm could determine on its own. We note that the results for NewDetKDecomp from Fischl et al. [14] are also such an “ensemble”, combining the results of three different GHD algorithms presented in [14]. Across all experiments, out of the 3648 instances in HyperBench, we have thus managed to solve over 2900. By “solved” we mean that the precise ghw could be determined in these cases. It is interesting to note that the Hybrid Algorithm on its own is almost as good as the “ensemble”. Indeed, the number of 2924 solved instances in case of the “ensemble” only mildly exceeds the the number of 2850 instances solved by the implementation of our Hybrid algorithm. The strength of the Hybrid algorithm stems from the fact that it combines the ability of the parallel Balanced Separator approach for quickly deriving lower bounds (i.e., detecting “Reject”-cases) with the ability of det- K -decomp for more quickly deriving upper bounds (i.e., detecting “Accept”-cases).

Figure 5 shows runtimes for all positive runs of the Hybrid algorithm over all instances of HyperBench with an increasing number of CPUs used, where the used width parameter ranges from 2 to 5. The blue dots signify the median times in milliseconds, and the orange bars show the number of instances which produced time-outs. We can see that increasing the CPUs either reduces the median (solving the same instances faster) or reduces the number of instances which timed out. Actually, reducing the number of time-outs is potentially a much higher speedup than

Table 5 Comparison of BalancedGo, HtdSMT and TULongo on the PACE 2019 Challenge, Track 2a

Method	# of solved instances	# of solved private instances	t_{avg} (sec)	t_{sum} (h)
BalancedGo	172	86	134.24	3.21
HtdSMT	165	80	128.67	2.89
TULongo [34]	70	38	105.58	1.11

Columns t_{avg} and t_{sum} show the average time and the total time, respectively, over all private instances

merely reducing the median, and also of higher practical interest, as it allows us to decompose more instances in realistic time. It should be noted that the increase of the median time when we go from 8 CPUs to 12 CPUs does not mean at all that the performance degrades due to the additional CPUs. The additional time consumption is solely due to the increased number of solved instances, which are typically hard ones. And the computation time needed to solve them enters the statistics only if the computation does not time out (Table 5).

In order to fully compare the strengths and weaknesses of each of the discussed decomposition methods, we also investigated the number of instances that could only be solved via a specific approach. This can be seen in Table 6. We see that while our approach clearly dominates this metric, there are still many cases where other methods were more effective.

In Fig. 6 we see an overview of the distribution of the ghw of all solved instances of our approach, and as a comparison we see how many instances for each width could be determined by NewDetKDecomp.

For the computationally most challenging instances of HyperBench, those of $ghw \geq 3$, our result signifies an increase of over 70 % in solved instances when compared with [14]. In addition, when considering the CSP instances from real world applications, we managed to solve 763 instances, almost doubling the number from NewDetKDecomp. In total, we now know the exact ghw of around 70% of all hypergraphs from CSP instances and the exact ghw of around 75% of all hypergraphs of HyperBench.

Another aspect of our solver we wanted to explore was the memory usage, and whether lifting the restriction to merely 1 GB of RAM makes a difference in the number of GHDs that can be found. We therefore looked at all test runs of lower width, ≤ 5 where our solver timed out. There were 91 such instances. This restriction seems justified as the width parameter affects the complexity of determining the ghw exponentially, thus it is only for lower widths that one would expect memory to

Table 6 Overview of exclusively solved instances of HyberBench for each decomposition method

Method	#exclusively solved
BalancedGo <i>ensemble</i>	284
NewDetKDecomp	11
HtdSMT	67

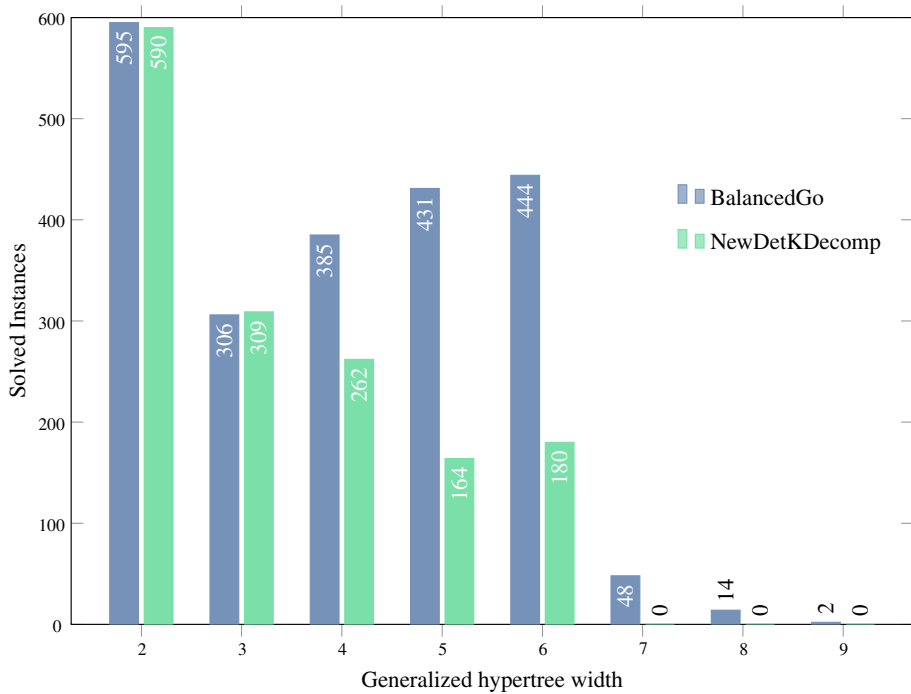


Fig. 6 Overview of the distribution of the *ghw* of the solved instances

become a limiting factor as opposed to time. We reran these 91 test instances using 24 GB of RAM. It turned out that the increase in available memory made no difference, however, as all 91 tests still timed out. In other words, the limiting factor in computing hypergraph decompositions is time, not space.

We stress that, in the first place, our data in Table 4 is not about time, but rather about the number of instances solved within the given time limit of 1 hour. And here we provide an improvement for these practical CSP instances of near 100% on the current state of the art; no such improvements have been achieved by other techniques recently. It is also noteworthy, that the Hybrid algorithm alone solved 2850 total cases, thus beating the total for NewDetKDecomp in [14], which, as mentioned, combines the results of three different GHD algorithms and also beating the total for HtdSMT [38].

Comparison with PACE 2019 Challenge In addition to experiments on HyperBench, we also compared our implementation with various solvers presented during the PACE 2019 Challenge [11], where one track consisted in solving the exact hypertree width. We took the 100 public and 100 private instances from the challenge (themselves a subset of HyperBench), and tried to compute the exact *ghw* of the instances within 1800 seconds, using at most 8 GB of RAM. Since our test machine is different

from the one used during PACE 2019 Challenge, we took the implementations of the winner and runner up, HtdSMT and TULongo [34] and reran them again using the same time and memory constraints. The results can be seen in Table 5. *BalancedGo* managed to compute 86 out of the 100 private instances, improving slightly on HtdSMT. It is noteworthy that this was accomplished while computing GHDs, instead of the simpler HDs which were asked for during the challenge.

6 Conclusion and outlook

We have presented several generally applicable algorithmic improvements for hypergraph decomposition algorithms and a novel parallel approach to computing GHDs. We have thus advanced the ability to compute GHDs of a significantly larger set of CSPs than previous GHD algorithms. This paves the way for more applications to use them to speed up the evaluation of CSP instances.

For future work, we envisage several lines of research: first, we want to further speed up the search for a first balanced separator as well as the search for a next balanced separator in case the first one does not lead to a successful decomposition. Note that for computing any λ -label of a node in a GHD of width $\leq k$, in principle, $O(n^{k+1})$ combinations of edges have to be investigated for $|E(H)| = n$. However, only a small fraction of these combinations is actually a balanced separator, leaving a lot of potential for speeding up the search for balanced separators. Apart from this important practical aspect, it would also be an interesting theoretical challenge to prove some useful upper bound on the ratio of balanced separators compared with the total number of possible combinations of up to k edges.

So far we were focused on the efficient and parallel computation of GHDs via balanced separators. It would be interesting to explore a similar approach for the computation of hypertree decompositions (HDs) [19]. The big advantage of HDs over GHDs is that their computation is tractable (for fixed k) even without adding certain subedges. On the other hand, HDs require the root of the decomposition to be fixed. In contrast, a GHD can be rooted at any node and the GHD algorithm via balanced separators crucially depends on the possibility of re-rooting subtrees of a decomposition. Significantly new ideas are required to avoid such re-rooting in case of HDs. First preliminary steps in this direction have already been made in [3] but many further steps are required yet.

In this work, we have looked at the decomposition of (the hypergraphs underlying) CSPs. The natural next step is to apply decompositions to actually solving CSP. Hence, another interesting goal for future research is harnessing Go's excellent cloud computing capabilities to extend our results beyond the computation of GHDs to actually evaluating large real-life CSPs in the cloud.

Appendix : A: Description of the sequential balanced separator algorithm

Algorithm 2 SequentialBalSep [14].

Input: A hypergraph H .
Parameter: An integer $k \geq 1$.
Output: A GHD of H of width $\leq k$ if it exists, NULL otherwise.

```

1 Main
2   Make  $H$  globally visible
3   return Decompose ( $H, \emptyset$ )

4 Function Decompose ( $H'$ : hypergraph,  $S_p$ : set of special edges)
5   if  $|E(H' \cup S_p)| == 1$  then
6     return node  $u$  with  $B_u \leftarrow V(H' \cup S_p)$  and  $\lambda_u \leftarrow E(H' \cup S_p)$ 
7   if  $|E(H' \cup S_p)| == 2$  then
8     Let  $e_1, e_2$  be the two edges of  $H' \cup S_p$ 
9     Create node  $u$  with  $B_u \leftarrow e_1$  and  $\lambda_u \leftarrow \{e_1\}$ 
10    Create node  $v$  with  $B_v \leftarrow e_2$  and  $\lambda_v \leftarrow \{e_2\}$ 
11    AttachChild ( $u, v$ )
12    return  $u$ ;

13  $BalSepIt \leftarrow \text{InitBalSepIterator}(H, H', S_p, k)$ 
14 while HasNext ( $BalSepIt$ ) do
15    $\lambda_u \leftarrow \text{Next}(BalSepIt)$ 
16    $B_u \leftarrow B(\lambda_u)$ 
17    $subDecomps \leftarrow \{\}$ 
18   foreach  $c \in \text{ComputeSubhypergraphs}(H', S_p, B_u)$  do
19      $\mathcal{D} \leftarrow \text{Decompose}(c.H, c.S_p)$ 
20     if  $\mathcal{D} \neq \text{NULL}$  then
21        $subDecomps \leftarrow subDecomps \cup \{\mathcal{D}\}$ 
22     else
23        $subDecomps \leftarrow \text{NULL}$ 
24     break
25   if  $subDecomps == \text{NULL}$  then
26     continue
27   return BuildGHD ( $B_u, \lambda_u, subDecomps$ )
28 return NULL

```

Acknowledgements Georg Gottlob is grateful to Cristina Zoltan for having suggested Go as an appropriate language for the parallelisation of computing hypertree decompositions.

Author contributions All authors contributed equally to this work.

Funding Open access funding provided by TU Wien (TUW). This work was funded by the Austrian Science Fund (FWF):P30930-N35. 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).

Data availability All the data used in our experimental evaluation has been published in an open access format (see [22]).

Code availability The source code of our implementation is publicly available under <https://github.com/cem-okulmus/BalancedGo>.

Declarations

Conflict of interests The authors have no conflicts of interest to declare that are relevant to the content of this article.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit <http://creativecommons.org/licenses/by/4.0/>.

References

1. Aberger, C. R., Tu, S., Olukotun, K., & Ré, C. (2016). Emptyheaded: A relational engine for graph processing. In *Proceedings of the 2016 international conference on management of data, SIGMOD conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, (pp. 431–446).
2. Adler, I., Gottlob, G., & Grohe, M. (2007). Hypertree width and related hypergraph invariants. *European Journal of Combinatorics*, 28(8), 2167–2181.
3. Akatov, D. (2010). *Exploiting parallelism in decomposition methods for constraint satisfaction*. UK: Ph.D. thesis, University of Oxford. <https://ora.ox.ac.uk/objects/uuid:30773f0c-9b53-4684-b1c4-2d20c2322edd>.
4. Amroun, K., Habbas, Z., & Aggoune-mtalaa, W. (2016). A compressed generalized hypertree decomposition-based solving technique for non-binary constraint satisfaction problems. *AI Comm.*, 29(2), 371–392.
5. Aref, M., ten Cate, B., Green, T. J., Kimelfeld, B., Olteanu, D., Pasalic, E., Veldhuizen, T. L., & Washburn, G. (2015). Design and implementation of the logicblox system. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, (pp. 1371–1382).
6. Baader, F., & Nipkow, T. (1998). *Term Rewriting and All That*. Cambridge: Cambridge University Press.
7. Bodlaender, H.L. (1996). A linear-time algorithm for finding tree-decompositions of small tree-width. *SIAM J. Comput.*, 25(6), 1305–1317. <https://doi.org/10.1137/S0097539793251219>.
8. Cohen, D. A., Jeavons, P., & Gyssens, M. (2008). A unified theory of structural tractability for constraint satisfaction problems. *Journal of Computer and System Sciences*, 74(5), 721–743.
9. Cox-Buday, K. (2017). Concurrency in Go: Tools and techniques for developers. “O'Reilly Media Inc.”.
10. Donovan, A. A. A., & Kernighan, B. W. (2015). *The Go programming language*. Addison-Wesley Professional.
11. Dzulfikar, M. A., Fichte, J. K., & Hecher, M. (2019). The PACE 2019 parameterized algorithms and computational experiments challenge: The fourth iteration (invited paper). In *14th international symposium on parameterized and exact computation, IPEC 2019, september 11-13, 2019, munich, germany*, (pp. 25:1–25:23).
12. Fagin, R. (1983). Degrees of acyclicity for hypergraphs and relational database schemes. *J. ACM*, 30(3), 514–550. <https://doi.org/10.1145/2402.322390>.
13. Fichte, J. K., Hecher, M., Lodha, N., & Szeider, S. (2018). An SMT approach to fractional hypertree width. In *Principles and practice of constraint programming - 24th international conference, CP 2018, lille, france, august 27-31, 2018, proceedings*, (pp. 109–127).
14. Fischl, W., Gottlob, G., Longo, D. M., & Pichler, R. (2021). HyperBench: A benchmark and tool for hypergraphs and empirical findings. *ACM J. Exp. Algorithmics* 26. <https://doi.org/10.1145/3440015>.

15. Fischl, W., Gottlob, G., & Pichler, R. (2018). General and fractional hypertree decompositions: Hard and easy cases. In *Proceedings of the 37th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, Houston, TX, USA, June 10-15, 2018*, (pp. 17–32).
16. Floyd, R. W. (1962). Algorithm 97: Shortest path. *Communications of the ACM*, 5(6), 345.
17. Gottlob, G., Hutle, M., & Wotawa, F. (2002). Combining hypertree, bicomplex, and hinge decomposition. In *Proceedings of the 15th European Conference on Artificial Intelligence, ECAI 2002*, (pp. 161–165). IOS Press.
18. Gottlob, G., Leone, N., & Scarcello, F. (2000). A comparison of structural CSP decomposition methods. *Artificial Intelligence*, 124(2), 243–282.
19. Gottlob, G., Leone, N., & Scarcello, F. (2002). Hypertree decompositions and tractable queries. *Journal of Computer and System Sciences*, 64(3), 579–627.
20. Gottlob, G., Miklós, Z., & Schwentick, T. (2009). Generalized hypertree decompositions: NP-hardness and tractable variants. *J. ACM*, 56(6), 30:1–30:32.
21. Gottlob, G., Okulmus, C., & Pichler, R. (2020). Fast and parallel decomposition of constraint satisfaction problems. In *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, IJCAI 2020*, (pp. 1155–1162). <https://doi.org/10.24963/ijcai.2020/161>.
22. Gottlob, G., Okulmus, C., & Pichler, R. (2021). Raw data on extended experiments for balancedgo zenodo. <https://doi.org/10.5281/zenodo.4411863>.
23. Gottlob, G., & Samer, M. (2008). A backtracking-based algorithm for hypertree decomposition. *ACM J. Exp. Algorithmics* 13. <https://doi.org/10.1145/1412228.1412229>.
24. Graham, M. H. (1979). On the universal relation. Tech. rep. University of Toronto.
25. Grohe, M., & Marx, D. (2014). Constraint solving via fractional edge covers. *ACM Trans. Algorithms*, 11(1), 4:1–4:20. <https://doi.org/10.1145/2636918>.
26. Gyssens, M., Jeavons, P., & Cohen, D. A. (1994). Decomposing constraint satisfaction problems using database techniques. *Artificial Intelligence*, 66 (1), 57–89.
27. Habbas, Z., Amroun, K., & Singer, D. (2015). A forward-checking algorithm based on a generalised hypertree decomposition for solving non-binary constraint satisfaction problems. *J. Exp. Theor. Artif. Intell.*, 27(5), 649–671.
28. Hoare, C. A. R. (1978). Communicating sequential processes. *Communications of the ACM*, 21(8), 666–677.
29. Kloks, T. (1994). Treewidth, computations and approximations. In *Lecture notes in computer science*, (vol. 842). Springer. <https://doi.org/10.1007/BFb0045375>.
30. Kolaitis, P.G., & Vardi, M.Y. (2000). Conjunctive-query containment and constraint satisfaction. *J. Comput. Syst. Sci.*, 61(2), 302–332. <https://doi.org/10.1006/jcss.2000.1713>.
31. Korhonen, T., Berg, J., & Jarvisalo, M. (2019). Enumerating potential maximal cliques via SAT and ASP. In S. Kraus (Ed.) *Proceedings of the twenty-eighth international joint conference on artificial intelligence, IJCAI 2019, Macao, China, August 10-16, 2019*, (pp. 1116–1122). <https://doi.org/10.24963/ijcai.2019/156>.
32. Lagergren, J. (1990). Efficient parallel algorithms for tree-decomposition and related problems. In *31st Annual symposium on foundations of computer science, St. Louis, Missouri, USA, October 22-24, 1990*, (Vol. I, pp. 173–182). IEEE Computer Society. <https://doi.org/10.1109/FSCS.1990.89536>.
33. Lalou, M., Habbas, Z., & Amroun, K. (2009). Solving hypertree structured CSP: sequential and parallel approaches. In *Proceedings of the 16th RCRA workshop on Experimental Evaluation of Algorithms for Solving Problems with Combinatorial Explosion, RCRA@AI*IA 2009, Reggio Emilia, Italy, December 11-12, 2009*, CEUR Workshop Proceedings, vol. 589. CEUR-WS.org.
34. Longo, D.M. (2019). Pace2019 hypertree width heuristic. <https://doi.org/10.5281/zenodo.3236369>.
35. Marx, D. (2013). Tractable hypergraph properties for constraint satisfaction and conjunctive queries. *J. ACM*, 60(6), 42:1–42:51.
36. Meiri, I., Pearl, J., & Dechter, R. (1990). Tree decomposition with applications to constraint processing. In *Proceedings of the 8th national conference on artificial intelligence, Boston, Massachusetts, USA, July 29 - August 3, 1990*, 2 Volumes, (pp. 10–16). AAAI Press / The MIT Press. <http://www.aaai.org/Library/AAAI/1990/aaai90-002.php>.
37. Robertson, N., & Seymour, P.D. (1986). Graph minors. II. algorithmic aspects of tree-width, (Vol. 7. [https://doi.org/10.1016/0196-6774\(86\)90023-4](https://doi.org/10.1016/0196-6774(86)90023-4).
38. Schidler, A., & Szeider, S. (2020). Computing optimal hypertree decompositions. In *Proceedings of the symposium on algorithm engineering and experiments, ALENEX 2020, Salt Lake City, UT, USA, January 5-6, 2020*, (pp. 1–11). SIAM. <https://doi.org/10.1137/1.9781611976007.1>.
39. Thain, D., Tannenbaum, T., & Livny, M. (2005). Distributed computing in practice: the Condor experience. *Concurrency - Practice and Experience*, 17(2-4), 323–356.
40. Warshall, S. (1962). A theorem on boolean matrices. *Journal of the ACM*, 9(1), 11–12.

41. Yannakakis, M. (1981). Algorithms for acyclic database schemes. In *Very large data bases, 7th international conference, September 9-11, 1981, Cannes, France, Proceedings*, (pp. 82–94).
42. Yu, C. T., & Özsoyoğlu, M. Z. (1979). An algorithm for tree-query membership of a distributed query. In *The IEEE computer society's third international computer software and applications conference, COMPSAC 1979, 6-8 November, 1979, Chicago, Illinois, USA*, (pp. 306–312).

Publisher's note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.