TU WIEN Informatics

Parallel Computation of Structural Decompositions

DISSERTATION

submitted in partial fulfillment of the requirements for the degree of

Doktor der Technischen Wissenschaften

by

Dipl.-Ing. Cem Okulmus, BSc
Registration Number 1215428

to the Faculty of Informatics

at the TU Wien

Advisor: Prof. Dr. Reinhard Pichler

The dissertation has been reviewed by:

Francesco Scarcello

Franz Wotawa

Vienna, 19th October, 2022

Cem Okulmus

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

Erklärung zur Verfassung der Arbeit

Dipl.-Ing. Cem Okulmus, BSc

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 19. Oktober 2022

Cem Okulmus

Acknowledgements

I would like to thank my advisor Reinhard Pichler for his mentorship during these four years of my PhD and for always being ready to discuss whatever strange ideas on hypergraph decomposition algorithms I had during this time. In this regards I feel especially indebted to Georg Gottlob for his constant stream of great ideas and proposals and infectious enthusiasm for research.

I would also like to thank my colleagues Matthias Lanzinger and Davide Longo, who enriched both as academic co-workers, with great ideas on all things hypergraphs, and as treasured friends.

Lastly, I want to thank the tireless support of my parents, without which I would have been able to reach this point.

Abstract

Identifying tractable problems and providing polynomial time algorithms for them has long been a key focus in the study of computational complexity. More recently, the focus has been on trying to identify tractable fragments of problems that are, in the general case, intractable. One such line of work is to look at the structural properties of problems such as constraint satisfaction and conjunctive query evaluation. The structure of these problems is naturally expressed as hypergraphs, and it is a long standing result that instances of these problems whose underlying hypergraph structure is acyclic are solvable in polynomial time. This then led to generalisations of acyclicity, in the form of hypergraph decompositions and the notion of *width* which signifies the complexity of the hypergraph that is being decomposed. Finding decompositions of low width, or determining quickly that none may exist, is thus of key importance for identifying and solving tractable fragments of constraint satisfaction problems or conjunctive query evaluation. In this thesis, we advance the ability to provide such low-width decompositions or proofs of their non-existence for these important problems.

As a first step, we focus on the particular type of hypergraph decomposition known as *generalized hypertree decompositions* (GHD). For this class, we provide the first parallel algorithm, called BalancedGo, that makes use of balanced separators. We also provide a number of general optimisations in the form of pre-processing steps and prove that they do not affect the correctness of any algorithm making use of them. We also show the potential of combining multiple distinct algorithms into one, and showcase the practicality of this by implementing a hybrid algorithm which combines our novel parallel algorithm with an older sequential one. This hybrid system turned to out to combine the best of both worlds, combining the strengths of each individual method. We conclude this first study with an extensive experimental evaluation of our developed methods against other state of the art methods for computing GHDs and show that our hybrid system outperforms all other approaches.

Following this, we explored another type of hypergraph decomposition, namely (regular) *hypertree decompositions* (HD). These represent a smaller class of hypergraph decompositions when compared to GHDs, which have the desirable property of being computable in polynomial time, when looking at decompositions of a fixed width. On the other hand, HDs are rooted, and our GHD algorithm constructs fragment of the GHD and heavily depends on re-rooting these fragments when combining them. In order to allow the use of balanced separators for constructing HDs, we provide an entirely new idea. This involved guessing pairs of nodes, parent node and child node. This allowed us to construct an HD in arbitrary order, allowing for the implementation of a parallel algorithm which can then use balanced separation to split the hypergraph into subproblems and work on them concurrently. We also describe how a series of optimisations allowed us to lessen the computational cost of the guess of pairs of nodes. We again conclude this line of study by an extensive experimental evaluation, this time focusing on state of the art decomposition methods for HDs and we show that our novel method log-k-decomp, as part of a more involved hybrid algorithm, managed to provide the best results.

Finally, we also explore the use of highly distributed systems for the computation of hypergraph decompositions. The focus on these distributed systems is motivated by two observations: the first one is the presence of very large hypergraphs in the commonly used benchmarks, which all existing state of the art methods have failed to solve optimally and the second one is the fact that the most computationally powerful systems are all highly distributed machines, composed of a large number of interconnected computers. Currently proposed methods cannot claim to make use of these distributed systems, as they need shared memory architectures, limiting them to be run on a single machine. By taking on some key ideas from the recent work of Gottlob, Lanzinger, Pichler and Razgon (JACM, 2021), we propose a novel algorithm, composed of several individual programs that can work concurrently. This distributed system is built on the notion of a block – which encodes subhypergraphs which need to be decomposed – and allows *workers* to independently search for balanced separators for all possible subhypergraphs that need to be explored to find a decomposition. These blocks are then processed at a central location to see if a decomposition of the input hypergraph has been found. This forms a distributed system and it has the property that the search for balanced separators can – in principle – use an unlimited number of machines and CPUs and thus tackle ever larger instances. We also develop a first prototype of this distributed algorithm and report on preliminary experimental results.

Kurzfassung

Das Identifizieren von praktisch lösbaren Problemen und das Entwickeln von konkreten Algorithmen die diese in Polynomialzeit lösen können ist ein Hauptaugenmerk der Komplexitätstheorie. In jüngerer Zeit lag der Schwerpunkt auf dem Versuch, praktisch lösbare Fragmente von Problemen zu identifizieren, die im allgemeinen Fall nicht praktisch lösbar sind. Eine solche Forschungsrichtung beschäftigt sich damit die strukturellen Eigenschaften von Problemen zu untersuchen, wie z. B. das Lösen eines Constraint-Satisfaction-Problems (CSP; deutsch: Bedingungserfüllungsproblem) oder das Auswerten einer Conjunctive Query (CQ; deutsch: konjunktive Anfrage). Die Struktur dieser Probleme wird in der Regel als Hypergraph ausgedrückt, und es ist ein bekanntes Ergebnis, dass Instanzen dieser Probleme, deren zugrunde liegende Hypergraphenstruktur azyklisch ist, in Polynomialzeit lösbar sind. In der Folge führte dies dann zu Werken die die Verallgemeinerung von Azyklizität untersuchten. Dies führte dann zu Hypergraph-Zerlegungen und dem Begriff von *width* (deutsch: Breite), der die Komplexität des zu zerlegenden Hypergraphen erfasst. Das Auffinden von Zerlegungen mit niedriger width oder das schnelle Bestimmen, dass keine solche Zerlegungen vorhanden sind, ist daher von entscheidender Bedeutung für das Identifizieren und Lösen effizient lösbarer Fragmente von CSP sowie die Auswertung von CQs. In dieser Dissertation ergründen wir neue Methoden um Zerlegungen mit geringer width effizient zu finden.

In einem ersten Schritt betrachten wir eine spezielle Art der Hypergraph-Zerlegung, die als *generalized hypertree decomposition* (GHD; deutsch: verallgemeinerte Hyperbaum-Zerlegung) bekannt ist. Für diese Klasse stellen wir den ersten parallelen Algorithmus bereit, genannt BalancedGo, welcher Resultate zu balancierte Separatoren aus der Graphtheorie verwendet. Wir entwickeln auch eine Reihe allgemeiner Optimierungen, die auf eine allgemeine Klasse von Algorithmen zur Berechnung von Hypergraph-Zerlegungen anwendbar sind, und wir beweisen, dass diese Optimierungen sicher zu verwenden sind, d. h. sie beeinträchtigen die Korrektheit der Algorithmen nicht. Wir zeigen auch das Potenzial, mehrere unterschiedliche Algorithmen zu einem zu kombinieren, und demonstrieren den praktischen Nutzen dieser Kombination, indem wir einen hybriden Algorithmus implementieren, der unseren neuartigen parallelen Algorithmus mit einem älteren sequentiellen kombinieren. Es stellte sich heraus, dass Hybridsysteme das Beste aus beiden Welten kombiniert und die Stärken der einzelnen Methoden vereinen. Wir beenden diese erste Studie mit einer umfangreichen experimentellen Evaluierung unserer entwickelten Methoden im Vergleich zu anderen Methoden aus der Literatur und zeigen, dass unser Hybridsystem alle anderen Ansätze schlagen kann.

Anschließend haben wir eine andere Art der Hypergraphen-Zerlegung untersucht, nämlich (reguläre) *hypertree decompositions* (HD; deutsch: Hyperbaum-Zerlegung). Diese stellen eine kleinere Klasse von Hypergraph-Zerlegungen im Vergleich zu GHDs dar, die die wünschenswerte Eigenschaft haben, in Polynomialzeit berechenbar zu sein, wenn Zerlegungen mit konstanter width betrachtet werden. Die Eigenschaften von HDs machen es schwierig, den gleichen Ansatz, den wir für GHDs verwendeten, direkt wieder zu verwenden. Stattdessen präsentieren wir hier eine völlig neue Idee. Dies beinhaltete das Erraten von Knotenpaaren in der finalen HD, je einen Elternknoten und je einen Kindknoten. Das Berücksichtigen aller dieser Kombinationen ermöglichte es uns, eine HD in beliebiger Reihenfolge zu konstruieren, was in der Folge die Implementierung eines parallelen Algorithmus ermöglichte, der dann balancierte Separatoren verwenden kann, um den Hypergraphen in Teilprobleme aufzuteilen und parallel an ihnen zu arbeiten. Wir beschreiben auch, wie wir durch eine Reihe von Optimierungen den Rechenaufwand für das Schätzen von Knotenpaaren verringern konnten. Wir schließen dieses Kapitel erneut mit einer experimentellen Evaluierung ab, diesmal mit Fokus auf moderne Methoden für das Berechnen von HDs, und wir zeigen, dass unsere neuartige Methode log-k-decomp, als Teil eines komplexeren Hybridalgorithmus, die besten Ergebnisse erzeugt.

Schließlich untersuchen wir auch die Verwendung verteilter Systeme zur Berechnung von Hypergraph-Zerlegungen. Der Fokus auf diese verteilten Systeme ist durch zwei Beobachtungen motiviert: die erste Beobachtung ist die Existenz von sehr großen Hypergraphen in den Benchmarks die für Hypergraph-Zerlegungen verwendet werden. Diese sehr schweren Instanzen konnten von keinem der bestehenden Methoden optimal gelöst werden. Die nächste Beobachtung ist die Tatsache, dass die rechen-stärksten Computersysteme in der Regel nicht aus einzelnen Maschinen bestehen, sondern aus einer großen Anzahl an verbundenen Maschinen. Die bisher vorgeschlagenen Methoden zur Hyperbaum-Zerlegung sind allerdings nur dafür gedacht, auf Einzelmaschinen zu laufen. In dem wir Schlüsselideen aus einer neuen Arbeit von Gottlob, Lanzinger, Pichler und Razgon (JACM, 2021) übernehmen, entwickeln wir einen neuartigen Algorithmen, welcher aus mehreren individuellen Programmen besteht welche simultan arbeiten können. Dieses verteilte System basiert auf der Idee von Blöcken – ein Block ist hier stellvertretend für einen konkreten Teilhypergraphen welcher zerlegt werden muss – und erlaubt es dass *Arbeiter* unabhängig von einander nach balancierten Separatoren für alle möglichen Teilhypergraphen suchen, welche untersucht werden müssen um eine Hypergraph-Zerlegung zu finden. Diese Blöcke werden dann an eine zentrale Stelle gesendet, wo dann festgestellt werden kann ob bereits genug Information vorhanden ist um eine vollständige Hypergraph-Zerlegung bestimmen zu können. In Summe formt dies ein verteiltes System und es hat die Eigenschaft dass die Suche nach balancierten Separatoren auf einer unbeschränkten Anzahl an Maschinen ausgeführt werden kann. Wir entwickeln als Teil dieser Arbeit auch einen ersten Prototypen von diesem verteilten Algorithmus und berichten von ersten experimentellen Ergebnissen.

Contents

Chapter 1

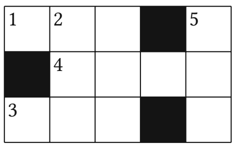
Introduction

In order to make this thesis approachable for a general audience, the first two sections of this introduction are written with somewhat less technical rigour, motivating instead the high-level ideas leading to the relevant research questions. Starting from Section 1.3, we provide a more technical description into the use of structural decomposition methods in the area of constraint satisfaction problems as well as conjunctive query evaluation. A reader that is more familiar with either of these topics can easily omit the initial sections and jump directly to Section 1.3.

The work presented here was created in close collaboration with many colleagues. Detailed acknowledgments and key publications for the research topics of this thesis are provided in Section 1.4.

1.1 Importance of Constraint Satisfaction Problems

Many problems we face in real life can be described as having a large set of possible choices one is faced with, while only having a relatively small set of actual solutions. One common example that many people likely face are scheduling problems of any kind. One needs to arrange a meeting, or a party, or any other kind of event where some number of people need to attend for some time. The problem lies in the fact that every person has some constraints. These can be things like their work hours, having to bring their children to school or other events they have already scheduled before. If not everyone can agree on the same date, we might also have to choose which combination of people we want to invite or we might also try changing the duration of the event. It is easy to see that as the number of participants grows and our options to modify parameters of the event, this problem becomes ever harder to solve. Another problem with similar properties can be found in many newspapers, the crossword puzzle. One is given a grid, with empty boxes and some black ones that are blocked. Each row and column has some hint, a word that is sought. The difficulty lies not only in finding words that have the right length and match the given clue, but also finding words that are consistent with the other solutions, where they cross in the grid. Thus, in addition to the potentially large number of possible words that satisfy the hint, we need to consider their combinations too. As this type of problem grows, the search space explodes in size. An example for a crossword puzzle can



Across	Down
1. deity	2. not young
3. song of praise	5. furniture for sleeping
4. body of water	

Figure 1.1: An example for a crossword puzzle. On the left is the grid where the solution needs to be provided, on the right are the definitions of the puzzles which need to be solved.

be seen in Figure 1.1. Finally, another type of problem common in real-life are configuration problems. When purchasing a new car, for example, there is a large number of options on additional features that the car could have. For each category, like the type of tyre, we are given a list of available choices. Between two separate categories, there might be interdependencies. One choice in one category might require the choice of another option in another category. By that same logic, two choices in separate categories might also exclude each other. As before, as the number of categories and options increases, the number of possible combinations rises too and in fact it rises exponentially. All these problems are known as *combinatorial problems* and in case we have some goal we want to minimise or maximise, one also speaks of combinatorial optimisation. CSPs also occur very frequently in Artificial Intelligence research [10, 28, 66, 76].

For all three examples, we could try to find solutions by testing every possible combination, essentially enumerating the entire search space. Indeed, for very small instances, such as the small crossword puzzles one finds in the newspaper, this might be a good idea, with little implementation overhead and giving us a solution in reasonable time. If we consider larger instances, however, this method, called a brute force method, will require more and more time as the search space grows, quickly reaching its limits. Without a similarly exponential growth in computational power, we would require far better algorithms to solve such complex instances of combinatorial problems. Due to the fact that many of these combinatorial problems are NP-hard [17], it is generally assumed that there is no way of finding an algorithm to solve them in polynomial time.

The formalism of constraint satisfaction problems (CSP) very naturally captures these types of problems, and despite the computational complexity we mentioned, the need to solve these problems in the real world has led to many highly sophisticated CSP solvers to be developed. These solvers are capable of solving many instances in reasonable time and continue to improve, as can be seen by various competitions. Despite the success of this area, however, there are still many hard instances that cannot be solved well. This is of course not surprising, as we are still talking about NP-hard problems. However, developments such as the rise of Big Data in the real world will likely only push the need to solve ever larger combinatorial problems, and thus there is a constant need to identify ever larger fragments that can be solved efficiently.

Our work will highlight one promising idea for how to speed up CSP solving, by the use of the structural properties of CSPs. By the structure, we refer to how things are connected. To return to the example of the crossword puzzle, we can already see the structure of the problem in its grid layout. Every word that needs to be filled in can, in principle, intersect with a large number of other words. It is these intersections that lead to a combinatorial explosion in the search space. These structural properties are therefore a critical component for the complexity of CSPs. Our work is now aimed at providing tools that allow for nearly all real world CSPs to quickly identify whether they are solvable in polynomial time due to such properties.

1.2 Importance of Conjunctive Queries

Another area that is perhaps even more important to every day life, especially in the commercial area, is the use of databases. Storing large amounts of structured data and being able to answer queries on that data in a short amount of time is crucial to our modern world. There is essentially no complex system which needs to work in any form with large amounts of information that is not reliant on this. For the sake of brevity and since it can be assumed that practically any reader will be familiar with the use cases for databases, we will omit any specific examples here.

For our work, we want to focus on a specific subset of queries that one can pose to a relational database management system (RDBMS). We will refer to this subset as *conjunctive queries* (CQs), and a more formal definition will follow later in this thesis. For the purposes of this introduction, it should suffice to claim that CQs correspond to simple SELECT-FROM-WHERE queries in SQL, where only equality between table attributes is allowed in the WHERE clause. To be more specific, we are also excluding the ability to form subexpressions within either clause, and we also exclude the use of any views. The latter is not a real restriction, however, since one can simply “unpack” the view, as long as the view itself is still a CQ, to form a larger expression.

It might be surprising that this limited subset of SQL is already NP-complete in combined complexity [11]. That is, unless P = NP, it is impossible to solve this type of problem efficiently. The reason for this lies in the fact that CQs allows us to encode combinations of potential solutions. To connect it with the crossword example in CSPs, we can use the individual tables to store the possible words for a given row or a given column, and then use join conditions to indicate where the solutions must overlap to satisfy the crossword instance.

In many systems, heuristics are used to tackle ever larger instances of CQs. As the number of tables that need to be joined increases, systems struggle to find effective query plans. The RDBMS PostgreSQL [75], for example, uses genetic algorithms to try to find some order of joins that will yield the best run time, when dealing with queries which involve a high number of tables. Unsurprisingly, these approaches have no runtime guarantees what so ever. A recent paper that explores this topic is from Mancini et al. [70]. They produce synthetic CQs over a real-world database, and show that existing RDBMSs have a very hard time dealing with queries that involve a larger number of tables, while their own research prototype for a parallel query engine is still able to solve a majority of them within milliseconds, even in cases where PostgreSQL times out.

It is notable that when one analyses the synthetic queries from [70], it turns out that a majority of them are acyclic. It is a long standing result in the literature that CQs, despite their complexity in the general case, are solvable in output-polynomial time when the structure of the instance is acyclic [93]. So despite being faced with tractable instances, modern RDBMSs do not achieve

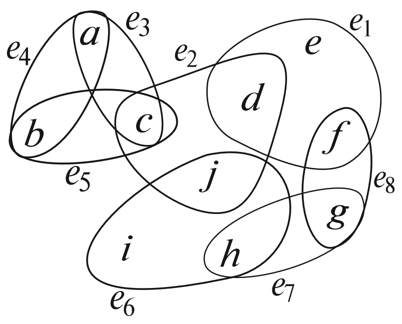


Figure 1.2: An example visualisation of a hypergraph.

competitive run times. Clearly the use of structural properties would be a natural idea. One of the core aims of our work is to give these systems better tools to identify acyclicity as well as generalisations of acyclicity very quickly, and also provide witnesses that show how a good query plan might look like in the form of hypergraph decompositions. While we do not believe that decompositions alone will suffice to enable RDBMSs to solve these CQs efficiently, as further quantitative information also needs to be considered, it is a first important step in this direction. We note again that our aim here is not to provide prototypes for the use of decompositions, but instead solve the more fundamental problem of producing decompositions. After all, without the ability in the first place to quickly produce good decompositions for a large number of real-word instances, the question of using them cannot have practical value.

1.3 Exploiting Generalisations of Acyclicity

When talking about the structure of CQs or CSPs, or really any kind of problem, it is sensible to use graphs, since they allow us to model only the structure – how various parts of the instances connect with each other – and abstract away all the other aspects of the problems. In the case of CQs, which correspond to simple SELECT-FROM-WHERE queries in SQL [3], we think of each table as a set and the attributes of the tables to be elements within this set. This gives us initially a disjoint set, as each table has distinct attributes. In the next step we look at the equalities in the WHERE clause and represent two attributes that appear inside an equation by only one element. Thus the sets will be connected by shared elements exactly as the corresponding tables are "connected" via their joins.

To express the structure of these sets, we make use of a generalisation of graphs which is called *hypergraphs*. Instead of edges only containing two vertices, we generalise the notion of edge to *hyperedge*, where a hyperedge can contain an arbitrary amount of vertices. Thus we can easily represent the interconnected set of sets we described above as a hypergraph, which is just a set of hyperedges. An example visualisation of a hypergraph can be seen in Figure 1.2. This hypergraph has eight hyperedges, and the visualisation shows the vertex sets that make up each hyperedges, for example e_6 consisting of the vertex set $\{h, i, j\}$.

While there is only a singular notion of acyclicity for regular graphs, there are multiple possible notions for acyclicity when looking at hypergraphs. For a thorough introduction to types of acyclicity in hypergraphs, we refer the interested reader to [21]. Throughout this thesis, we shall use the notion of α-acyclicity when talking about the acyclicity of hypergraphs. It has been shown by Yannakakis [93] that solving CQs is tractable when they exhibit an acyclic structure. This is made more precise by Gottlob, Leone and Scarcello [41] who show that this is evaluating acyclic CQs is complete for the complexity class LogCFL. α-acyclicity has long been known to lead to many useful applications in database theory [9, 94]. Since the two problems of CSPs and CQs can be reduced to each other, the same algorithm gives rise to a polynomial-time algorithm for solving CSPs as well.

So while the problems of CSP solving and CQ evaluation are tractable when we deal with acyclic instances, this does not immediately help us when dealing with cyclic cases. Especially for CSPs, it is not at all unusual to have cycles.

When looking at regular graphs, a very common generalisation of acyclicity is the use of tree decompositions (TD) [80, 81]. We formally introduce this concept in the next chapter. Since connected, acyclic regular graphs are simply trees, TDs measure the "closeness" of a graph to tree. A tree decomposition is a tree where the nodes of the tree have a *bag*, which is a set of vertices from the graph. For every edge of the graph, there needs to be a node whose bag contains all its vertices. Furthermore, when looking at all nodes of the TD that contain a certain vertex in their bag, then this subset of nodes must form a connected subtree within the tree decomposition. For a given tree decomposition, its width is simply the size of the largest bag minus one. And the *treewidth* of a graph is smallest width across all its tree decompositions. The closeness of a graph to a tree is expressed by its treewidth, where a treewidth of 1 indicates that we are dealing with trees, and the higher the width gets, the more "cyclicity" the graph exhibits. For a class of graphs, we say it has "bounded treewidth" if there exists a constant such that no graph in the class has a treewidth higher than this constant.

To use TDs for hypergraphs, we simply adapt the first property and define analogously that for TDs of hypergraphs it must hold that for each hyperedge of the hypergraph there needs to exist one node in the TD such that its bag contains all vertices of the hyperedge in question. The problem with TDs when using them for hypergraphs lies in the fact that the width of TDs is dependent on the size of the hyperedges covered in the bags. For regular graphs this is not a problem as all edges have the same size. In hypergraphs, this means that even classes of α-acyclic hypergraphs may have an unbounded treewidth: simply look at hypergraphs that consist of a single hyperedge, of unbounded size. Thus TDs are not a useful tool if we want to find islands of tractability for CQ evaluation and CSP solving based on their hypergraph structure.

The solution came in the form of *hypergraph decompositions*. To the best of our knowledge, the first of these were *query decompositions*, introduced by Chekuri and Rajaraman [14]. We shall not focus on them in this work, and will instead focus on the notion of hypertree decomposition from Gottlob, Leone and Scarcello [43]. There are two reasons for this. The first reason is that for a given hypergraph and positive integer k, computing query decompositions is intractable, even when only looking at query decompositions of width at most k. On the other hand, this problem is tractable for hypertree decompositions. The second reason is that query decompositions cannot permit lower widths for a given hypergraph when comparing them to hypertree decompositions, as is shown in [43]. Hypertree decompositions are extensions of TDs which additionally also have an edge cover. Thus in addition to the bag, each node of a hypertree decomposition has an edge cover too, which is a set of hyperedges and it is required that the union of the vertices of the hyperedges is a superset of the bag. The width of a hypertree decomposition is the size of the largest edge cover. Thus the width measure is no longer about the number of vertices of the involved edges, but instead directly about the number of edges needed to cover the vertices of the bag. We identify as the *hypertree width* (*hw*) of a hypergraph the smallest width of all its hypertree decompositions. Hypertree decompositions actually properly generalise α-acyclicity. All acyclic hypergraphs are identified by the fact that they have a *hw* of 1. To more formally explain why we focus on hypertree decompositions, we need to introduce the formal problem that our work will focus on solving.

CheckHD	
Input	hypergraph $H = (V, E)$;
Parameter	k;
Output	*HD* of H of width $\leq k$ if it exists and answer ‘no’ otherwise.

As we said before, the crucial property of hypertree decompositions is the fact that the CheckHD problem can be solved in polynomial-time. We say a family of hypergraphs has *bounded hypertree width*, if there exists some constant k such that every hypergraph in this family has *hw* of less or equal k. It has been shown that many results that hold on acyclic CQs or CSP can be naturally generalised for CQs or CSPs with bounded hypertree width [79]. There are also results in the literature that show the application of hypertree width in areas beyond CQs and CSPs [31, 32, 33]. There have since also been further generalisations, such as *generalized hypertree decompositions* [4], *fractional hypertree decompositions* [53] and when focusing on width measures alone, even further generalisations such as adaptive width [72] or submodular width [73]. For the purposes of this thesis, we will restrict our focus on only two classes of decompositions, generalized hypertree decompositions and (regular) hypertree decompositions.

The use of hypergraph decompositions has also found its way into experimental database systems or CSP solvers [1, 2, 5, 7, 56, 67, 77, 90], as well as commercial systems [59, 60, 61, 62]. These research prototypes and early commercial systems show the potential of hypergraph decomposition for the areas of CQ evaluation and CSP solving.

There have also been initial attempts at computing hypergraph decompositions [23, 25, 50, 64, 74, 86]. When looking at comprehensive data which compares their performance against real-world datasets of hypergraphs, however, these first attempts show their limitations. Thus, while we are given a promising solution to checking structural properties and providing witnesses in the form of hypergraph decompositions, there is still more work to do to make this useable in practice for a large class of instances of CSPs and CQs. This is exactly where this thesis picks up, aiming to provide significant improvements to the practical computation of various hypergraph decomposition methods.

In order to measure the effectiveness of any hypergraph decomposition method, there needs to be a benchmark of hypergraphs, representing a diverse sample of possible hypergraph structures. Such a dataset was provided by Fischl, Gottlob, Longo and Pichler [39] in the form of HyperBench. It consists of 3648 hypergraph instances, from a large variety of sources. This dataset features hypergraphs of real-world instances of CSPs and CQs, as well as synthetic instances. It contains a diverse set of hypergraphs and therefore provides a meaningful benchmark for the effectiveness of a given decomposition method in practice. HyperBench has since been used many times in the literature for the purpose of experimental evaluation of decomposition methods [20,37,49,85,86].

Combinatorial top-down hypergraph decomposition. Before we list our contributions, we must first introduce some key notions that make up the type of algorithm we explored and designed in our work. The first such algorithm was designed by Gottlob, Leone and Scarcello [43]. This was a theoretical algorithm which was meant to run on a generalisation of Turing Machines called *Alternating Turing Machines* [12]. As such, it is not suitable for a direct implementation in real-world machines. The two core ideas of *combinatorial* guess of edge covers, and a *top-down* construction of a hypergraph decomposition, were already present. The first deterministic algorithm, which replaced the mechanism of guessing among a set of combinations with a backtracking-based search, was provided by Gottlob and Samer [50], called det-k-decomp. The backtracking is necessary to ensure that in the search for a decomposition, all possible choices for a node and all possible choices for building a decomposition by combinations of nodes are accounted for. The soundness and, of particular importance, the completeness of this algorithm was also shown by the authors. In the sequel, we will refer to these concepts, namely combinatorial, backtracking-based, top-down algorithms for hypergraph decomposition.

1.4 Main Challenges and Contributions

We have so far highlighted the potential of hypergraph decompositions for the tasks of solving CSPs and the evaluation of CQs, as they generalise the useful properties of acyclicity for a larger class of instances. They thus bring very desirable complexity guarantees for these instances and the use of decompositions would extend the ability of existing CSP solvers and RDBMS to solve more effectively a larger class of instances. Nonetheless, to achieve this goal the ability to provide hypergraph decompositions for real-world instances needs to be improved. We present here a number of research questions and challenges related to this endeavour and answer some of these questions as well as tackle some of the involved challenges.

The contributions we detail in this thesis were only possible through the joint work of multiple collaborators and the actual list of collaborators as well as key publications will be listed in the respective chapters.

1.4.1 The Fast and Parallel Computation of Generalized Hypertree Decompositions

The work presented as part of this subsection was achieved in close collaboration with Georg Gottlob and Reinhard Pichler. The main results were first published at the International Join

Conference for Artificial Intelligence (IJCAI) 2020 [47]. Initial work on this topic was also published at the Alberto Mendelzon International Workshop on Foundations of Data Management in 2019 [46]. An extended version of the first publication was subsequently submitted to the journal Constraints, and accepted in 2022 [49].

When considering the problem of computing GHDs and aiming to extend the ability to compute optimal decompositions for a wider class of instances within a given time budget, one observation is that none of the previously existing state of the art decomposition methods for GHDs utilised parallelisation. As our aim must be to utilise existing hardware to the best possible extent, it is apparent that purely sequential solutions cannot actually claim to do so. Aside from small embedded systems, virtually all computing hardware has multi-core CPUs. Thus it seems natural to ask if we can make use of parallelisation when tackling the problem of GHD computation.

Research Challenge: Are there effective parallel algorithms for the computation of generalized hypertree decompositions?

To give more context for why this is a complex problem, we shall highlight some key issues that effective combinatorial algorithms for computing GHDs need to address to meet this challenge.

Minimising synchronisation delay as much as possible As we are ultimately talking about search problems, any parallel implementation will need to split the search space into multiple parts. To achieve any kind of improvement in the time needed to find a solution, the implementation needs to ensure that any effort needed to synchronise parts of the search does not cause so much delay that any gains by the splitting of the search space are offset.

Finding a way to partition the search space equally among CPUs, and thus utilising the resources optimally As the aim is to use modern hardware optimally, it is also critical to design parallel algorithms for GHD computation which ensure that the work is split equally among the various CPU cores. This is needed to make sure that we can avoid idle times where one or more cores do not have any work to do, thus wasting resources.

Supporting efficient backtracking, a key element of all structural decomposition algorithms presented so far As we will explain when more formally introducing top-down approaches for computing hypergraph decompositions, backtracking is a key part of all of them. In addition to the two challenges above, a parallel algorithm also needs to ensure that backtracking is supported. This includes that no work must be lost when restarting the search and trying to find another possible solution, made harder for a parallel search split up among multiple cores.

Main Result 1: We have designed a novel parallel algorithm for the computation of GHDs and provide also an implementation using the programming language Go.

We were able to meet this challenge, in part due to utilising the modern programming language Go, designed by a team at Google in 2009. Among its many features, it implements support for concurrency via the use of so-called “goroutines”, themselves strongly inspired by the concept of *communicating sequential processes* [57]. Go provided us with the tools needed to implement a parallel algorithm for the computation of GHDs. The basis for this parallel implementation was the balanced separator algorithm introduced in [25].

While the use of parallelisation provided us a major speed-up, there were many cases that were still not solvable within our set time budget when run on our test machines. As a next step, we thus considered if there was still room to improve the implementation of top-down algorithm as a whole, looking beyond just our particular implementation. This led to the next challenge.

> **Research Challenge:** Are there general improvements for the computation of hypergraph decompositions?

For this challenge, while also aiming to provide optimisations which are specific to GHDs, we also wanted to systematically explore whether there are general ways of speeding up algorithms for hypergraph decomposition. We thus worked out a number of optimisations, consisting of the preprocessing of hypergraphs to reduce the effective search space. In addition to this, we also provide ways to speed up the search for balanced separators in the setting of GHDs via a more intelligent use of assumptions like the bounded intersection property (BIP), to be detailed in the next chapter.

> **Main Result 2:** We provide a number of algorithmic optimisations and techniques to speed up the computation of GHDs, applicable even beyond our research prototype.

Even after this step, we still noticed that the performance of many of the individual approaches was unsatisfactory. We did observe, however, that the strength and weaknesses of individual methods, including our parallel implementation was often complimentary, meaning that we observed cases where one algorithm was very effective, whereas the other was slow and cases where we saw the opposite behaviour. This led us to the next immediate question, namely to find ways to pool their various strengths together, while making up for the shortcomings of individual cases.

> **Research Challenge:** Is it possible to push the ability to compute GHDs for a large number of real world instances via the combination of multiple distinct algorithms?

We ended up choosing two methods. The first is the parallel balanced separator algorithm we developed as part of the earlier challenge. It proved effective when dealing with instances with very large search spaces, and was good at quickly determining when one was dealing with a negative instance. The other method is based on the existing algorithm det-k-decomp, introduced by Gottlob and Samer [50] and extended for GHD computation by Fischl et al. [25]. It is sequential in nature and is highly efficient for very small instances. Our combined hybrid thus initial uses the parallel Balanced Separator Approach to prune the search space, leading to smaller and smaller instances. Then it switches to det-k-decomp to solve the remaining small subproblems. This work also required us to completely rewrite det-k-decomp in the Go language, as we needed a tight integration to minimise any potential overhead from using two approaches at once.

Main Result 3: We propose a hybrid algorithm, combining our novel parallel algorithm with existing sequential methods, which outperforms either individual technique and show its effectiveness in an extensive empirical evaluation.

As part of this work, we thus also provide an experimental evaluation using the hybrid methods, as well as the involved individual methods and other state of the art decomposition methods. We show that the hybrid algorithm clearly outperforms all other methods, managing to optimally solve over 78% of all instances of the HyperBench for the problem of finding GHDs.

1.4.2 The Fast and Parallel Computation of Hypertree Decompositions

The work presented as part of this subsection was achieved in close collaboration with Georg Gottlob, Mathias Lanzinger and Reinhard Pichler. The main results were presented at the SIGMOD-SIGACT-SIGAI Symposium on the Principles of Database Systems (PODS) 2022 [37]. The journal ACM Transactions on Database Systems (TODS) also invited us to publish an updated version of this work as one of four "best of PODS 2022" papers.

The next topic in this thesis concerns the faster computation of a more restricted class of hypergraph decompositions, namely Hypertree Decompositions. There is a number of interesting properties they have, when compared to the wider class of GHDs. For one, finding a small HD, formally solving the check problem for a fixed constant width, is tractable for HDs, whereas in the general case the check problem for GHDs is NP-complete, already at width 2 [39, 45]. Another very interesting property of HD computation is the fact that it falls under the complexity class LogCFL [43, 91]. This class is known to be in principle highly parallelisable due to its connection to the boolean circuit classes AC^1 and NC^2 [42, 82, 83]. For more information on complexity classes for parallel computation models, we refer to Greenlaw and Hoover [52].

Given the effectiveness of the Balanced Separator Approach for computing GHDs in parallel, this leads us to the natural next challenge.

Research Challenge: Is it possible to use the Balanced Separator Approach beyond just the computation of Generalized Hypertree Decompositions?

What makes this challenge interesting, is the fact that HDs are actually rooted. That is, the order of nodes in an HD is important, and if we try to re-use the idea of "rerooting" – reversing the parent-child relationship inside the tree to form a new tree – we might end up with a result that is no longer a valid HD. Therefore, the use of balanced separators to split the problem into ever smaller subproblems cannot be directly used for HDs.

We took inspiration for this problem from an algorithm called divide-k-decomp, described in the PhD dissertation of Dimitri Akatov [6]. At first glance, it seems to describe a way of solving this exact problem, using balanced separation to compute HDs via a parallel algorithm. Sadly, when trying to implement this method and carefully analysing its informal description, we discovered that divide-k-decomp is not actually a correct algorithm for computing HDs. In the process of trying to understand what went wrong there, we actually came up with a novel idea to solve this challenge.

> **Main Result 4:** We propose a novel parallel algorithm for HDs, which allows the use of balanced separators for computing (regular) Hypertree Decompositions.

We shall proceed to briefly sketch out our solution. To utilise the Balanced Separator Approach for HDs, we need to determine the relative position of the subproblems to the final HD we are computing. In other words, we need to guess which subproblems end up being covered below the current node in the tree, and which belong further up. This ensures that we can carefully maintain the properties of a valid HD while still creating ever smaller subproblems which can be solved in parallel. The crucial idea, however, lies in the fact that to do so, we cannot just guess the edge covers of a single node, as we do in the GHD case. We instead guess edge covers of two nodes at once, the current node as well as its potential parent node in the tree. Making use of a critical Lemma from [43], combining the two lambda labels allows us to determine the complete bag of the current node, thus giving us the ability to determine the subproblems needed to construct the rest of the HD.

Our initial description of this algorithm, which first guesses a parent node and afterwards guesses a child node, while interesting from an algorithmic point of view, is too slow to help with finding a competitive decomposition method for HD solving. The number of possible parent-child parings quickly grows out of control for even moderate hypergraphs and edge cover sizes, leading to unsatisfactory run times. We were thus faced with the problem of finding ways to mitigate this cost.

> **Research Challenge:** Can the cost of guessing two nodes at once be mitigated, to allow for the design of a competitive algorithm?

We ended up with an optimised version of our algorithm, called log-k-decomp. In addition to many minor changes, one of the larger modifications lies in the idea of first looking for the target node, and trying to find the parent only afterwards. The reason this is faster, is that we can look for balanced separators right away. Since for most hard instances, there are only few possible choices of edges that lead to balanced separation, this leads to a reduction in the search space. Another key idea was to forgo computing the root first, allowing us to jump right into the problem of finding balanced separators.

Main Result 5: We propose a series of optimisations which enable us to to match and even exceed the performance of other state of the art decomposition methods for HDs.

As with our work for GHDs, we show the strength of our method by an experimental evaluation of various state of the art methods for computing HDs and show that log-k-decomp outperforms them all. We note that our final solution again employed a hybrid system, one that switches to using the far simpler algorithm det-k-decomp, once the subproblems have been reduced by a predefined metaparameter. As such the nature of the hybridisation here is more complex than in our previous work, where the switch to det-k-decomp depended only on the depth of the recursion.

1.4.3 Utilising Distributed Systems for the Highly Parallelised Computation of Hypergraph Decompositions

What follows is currently on-going work in collaboration with Matthias Lanzinger.

For the last topic of this thesis, we focus on the problem of finding ways of computing hypergraph decompositions with essentially unlimited levels of parallelisation. For the hardest instances in the HyperBench dataset, all existing approaches failed to find optimal decompositions. We also note that all existing systems so far run on single machines. Most commodity hardware only has CPUs with a relatively small number of cores, somewhere between 4 to 16. Even the most expensive server-grade CPUs have on a single machine little more than 100 cores. On the other hand, the most powerful computing systems today are clusters that pool the computational power of a large number of individual machines. It is therefore natural to ask if we could use these highly distributed computing systems for the purposes of computing hypergraph decompositions.

Research Challenge: Are there effective algorithms for computing hypergraph decompositions on highly distributed systems, such as modern commercial cloud platforms?

To highlight the complexity of this task, we consider that the two novel methods we presented so far, one for GHD and another for HD, require shared-memory systems and employ direct back-tracking to find new edge covers in case negative subproblems were encountered. A naive translation of such an algorithm to a distributed system, where the various recursive function calls are executed on different machines, would require a lot of back and forth communication, essentially leading to a huge communication cost. When dealing with highly distributed systems, communication cost is of key importance and needs to be minimised as much as possible if one wants competitive solutions.

Instead of trying to modify existing algorithms which were written with shared-memory systems in mind, we therefore look at an entirely new approach, better suited for this new scenario. We focus on the concept of *candidate tree decompositions* (CTD), described by Gottlob et al. [39].

There, a very high-level description of a polynomial time algorithm for the computation of CTDs is provided. While this algorithm is not at all suitable for a practical implementation, due to the fact that it requires all possible bags which might occur in any decomposition to be provided as input, it does highlight how the various subproblems that make up the combinatorial top-down approach can be encoded in the form of so-called “blocks”. A block is a pair of two vertex sets, where the *head* represents a bag, and the *tail* represents a connected component of the head when separating the input hypergraph. A block may be “satisfied”, if there exists a CTD such that its root has as its bag the head of the block, and the hypergraph covered by the CTD corresponds to the component in the tail.

Equipped with these ideas, we set out to implement our distributed algorithm, which does not actually require all bags to be provided as input and allows for an essentially unlimited number of machines to be used as part of the search for new blocks. We limit our search to blocks that have as head a balanced separator.

Main Result 6: Based on the idea of candidate tree decompositions from Gottlob et al. [39], we propose the first algorithm for computing GHDs on distributed systems.

We provide a pseudo-code description of our algorithm later in this thesis. We stress that this is an entirely new approach, with the algorithm from [39] only serving as the initial inspiration.

As part of our on-going work, we have also created a first prototype of this algorithm, written in the programming language Go and running on the Google Cloud Platform. We conclude this topic with a report on first promising results.

1.4.4 Exploring Threats to Validity of the Empirical Evaluations

We want to address here threats to validity [16, 19, 22] with respect to the empirical evaluations that will be presented in Chapters 3, 4 and 5. Since these chapters share very similar test methodology, it seems easier to summarise the salient points here in a single location.

Conclusion Validity. In [16], the authors define *conclusion validity* as follows:

Every empirical study establishes relationships between the treatment, represented by the independent variables, and the outcomes, represented by the dependent variables. The researcher derives conclusions from these relationships, which should have practical use. Conclusion validity refers to the belief in the ability to derive conclusions from the relationships.

In our case, the independent variables are the algorithms we want to consider and compare. The dependent variables the run times their implementations produce when run against the benchmark we consider. Hence, a threat to conclusion validity is an element of the design of the experiments that makes it difficult to draw such a conclusion from algorithms to run times. In our case, it could stem from using a sub-par implementation, thus making it hard to draw a conclusion on the usefulness of the underlying algorithm. We believe this threat is reasonably well addressed by referring to implementations developed by the respective authors of the algorithms themselves, who are surely well versed with their own work and make sure to publish only the best possible implementation, within their abilities. Of course, one could always potentially improve on an implementation with newer technology, newer techniques or exploring alternative programming languages. Thus this threat cannot be completely avoided, it can only be mitigated by making sure the best known implementations of the investigated algorithms are used under ideal circumstances (i.e. with enough CPU, memory and other needed resources) in order to make sure they run as effectively as possible.

Internal Validity. In [16], *internal validity* is defined as:

> *This validity refers to the belief that the changes to the dependent variable A are solely caused by changes of the independent variable set S of the model.*

Before we explain further, we want to differentiate internal validity from conclusion validity. Conclusion validity is the general claim, that there is a *relationship* between the dependant variables and the independent variables, and if we have such a relationship then it allows us to draw valid conclusions on the algorithms (i.e. the independent variables) by looking at run times of implementations (i.e. the dependent variables). Internal Validity is concerned with a finer point, namely that changes to dependent variables are *solely* caused by and due to the independent variables. Without conclusion validity, the question of internal validity is meaningless. On the hand, faulty internal validity could still threaten our ability to draw useful conclusions, even if we know there is a direct relationship between algorithms and run times of implementations.

So to repeat, the dependent variable we measure in all experiments of this thesis is the run time. And indeed we want to make the claim that the run times we measure are *solely* due to inherent strengths or weaknesses of the algorithms themselves. Threats to internal validity are thus factors that might affect the run times. This could be due to problems with our test machines, leading to run times being overly affected by the machines they run on, rather than the characteristics of the algorithms themselves, irrespective of the machine. As with the conclusion validity, these threats cannot be completely avoided, it can only be mitigated. In order to mitigate small variances in the run times introduced by the test machines, we avoid reporting on individual instances when tests include instances with low run times (below 1 second), and instead report on aggregates over 3000 of test cases. When conducting experiments, we also make sure that no other unrelated processes are running on a test machine, thus hopefully providing more reliable results for the specific implementations we test.

Construct Validity. In [16], *construct validity* is defined as:

> *Construct validity refers to the belief that the dependent variables and independent variables represent the theoretical concept of the phenomenon accurately.*

Since we focus here only on the computation of hypergraph decompositions, there is no external "phenomenon" we need to model or deal with. Thus, due to the theoretical nature of this work this particular definition of validity does not seem to apply, and as such we believe that we do not need to worry about threats to construct validity as well.

External Validity. Lastly, in [16], *external validity* is defined as:

> *External validity refers to the generalization of the results, e.g., it being "safe" to apply the results of a software study to all software of that type.*

In our context, this means whether our experiments on the chosen benchmark are truly representative of the kinds of hypergraph instances one would find when trying to find decompositions of the hypergraphs underlying real-world CQs and CSPs. This risk has been mitigated by the careful design of the HyperBench dataset itself, which draws not only from actual real-world samples of CQs and CSPs, but has also been enriched by synthetic instances [25]. Together, it seems likely that a rather large percentage of hypergraph structures one would find in the real-world has been covered. Of course, there is no way to prove this without in-depth real-world studies, which go beyond the scope of our work. As such, and like the other three types of risk, we cannot fully avoid some threats to external validity either.

On the role of alternative frameworks for parallelisation and distributed computing. As a final note on the validity of our experiments, we comment on a somewhat related topic, namely the chosen technologies with which we have realised our proof-of-concept implementations. We stress here that the focus of this thesis is not to produce implementations that can already be used inside real-world applications. Instead, we wish to showcase the usefulness of the underlying parallel and distributed algorithms we designed as part of this thesis. As such, we are already "happy" if we can show improvements on the state of the art with our proof-of-concept implementations. Nevertheless, one may ask if other technologies, such as the OpenMP [13] framework for parallel computation might have been used in Chapter 3 and Chapter 4 instead of the Go programming language [18]. For Chapter 5, there is then also the question if the MPI framework [54] for distributed computing could have been used, instead of extending Go with PubSub to enable message passing. We would answer these two questions positively. Indeed, for our implementations, we had to simply choose among a large field of options. We leave the exploration of other methods as future work, and while alternatives might have proven even more effective, improving on the run times we report in this thesis, this would only strengthen our results, namely the usefulness of the parallel and distributed algorithms we present.

1.5 Overview of this Thesis

The rest of this thesis is organised as follows. In Chapter 2 we recall needed definitions and results from the literature.

The following three chapters make up the main results of this thesis, where each chapter corresponds to one of the topics that was detailed in Section 1.4. While each of these topics follows a different goal in the search for better methods for computing hypergraph decompositions, it is recommended to read them in order since the overall aims connect very naturally. Each chapter also has its own introduction and conclusion, to provide context for how the work described fits into the overall scientific context.

The conclusion of the thesis is in Chapter 6. It provides a summary over all main results and puts them into the larger context, how they connect to other important subjects in the field. We also include a section dedicated to future work. In this section we list further research challenges and sketch out some possible new lines of research in the area of structural decomposition methods.

Chapter 2

Preliminary Definitions

In this chapter we provide the reader with the necessary definitions and concepts to follow the subsequent chapters of the thesis more easily. We also fix the notations that will be used throughout this work. Important results from the literature that are of general relevance to the entire thesis will also be stated in this chapter, whereas results with more specific relevance to individual chapters will be stated in those places where they play an important role. For an overall more in-depth introduction to the relevant definitions and topics in the area of hypergraph decompositions, we refer to [35].

2.1 CSPs, CQs & 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 $R_i = \{t_0 \dots, t_\ell\}$ a constraint relation which contains tuples of size n using values from a domain D. The variables of the constraints of P may overlap. A solution to P is an assignment $\phi : \bigcup_i S_i \to D$ from the variables to values from the domain D, such that for each constraint (S_i, R_i) with $S_i = \{s_0, \dots s_n\}$ and $R_i = \{t_0 \dots, t_\ell\}$, we have that there exists some $t \in R_i$ such that $\{\phi(s_0), \dots, \phi(s_n)\} = t$. Less formally, for any constraint (S_i, R_i) in P the solution ϕ must assign the constraint variables S_i to some tuple in the constraint relation R_i. For the standard definitions and more in-depth discussion of CSPs, we refer the interested reader to the book on this topic by Tsang [89].

Conjunctive queries (CQs) are arguably one of the most fundamental types of queries in the database world. Formally, they are given by a first-order formula ϕ using only the connectives in $\{\exists, \wedge\}$ and disallowing $\{\forall, \vee, \neg\}$. A *relational signature* σ is a finite set of relation symbols, each with some arity. A database D over a relational signature σ consists of a finite domain and a relation R^D for each relation symbol R in the signature. A CQ q is a set of positive literals, each from a relational signature σ:

$$q : \mathrm{OUT}(\mathbf{y}) :- R_i(\mathbf{x}_0) \wedge \cdots \wedge R_\ell(\mathbf{x}_\ell).$$

The output literal OUT is merely syntactic sugar and used here for notational convenience.

In FO, the output variables would correspond to free variables and all other variables would be existentially quantified. In our notation above, we require that the output variables are a subset of the variables occurring in literals, or formally $\mathbf{y} \subseteq \bigcup_i \mathbf{x}_i$. For a CQ q, we use the shorthand $\text{vars}(q) = \bigcup_i \mathbf{x}_i$. An *assignment* $a : \text{vars}(q) \to D$ is a total function that assigns to each variable of q some value of the domain. Given a signature σ and a database D over σ, we say an assignment a *satisfies* a given CQ q, if for each literal $R_i(\mathbf{x}_i)$, with $\mathbf{x_i} = \{x_0, \dots, x_n\}$, we have that $\{a(x_0), \dots, a(x_n)\} \in R^D$. Given a satisfying assignment a of a CQ q, we say the *output* of q under a is $\text{OUT}_a = \{a(y_0), \dots, a(y_m)\}$, where the set $\{y_0, \dots, y_m\} = \mathbf{y}$ consists of the output variables of q. To *evaluate* a CQ q means to iterate over the set of all satisfying assignment functions A, and producing the union of outputs:

$$\mathbf{Eval}(q) = \{\text{OUT}_a \mid a \in A\}.$$

We can observe that CQs correspond to SELECT-FROM-WHERE queries in SQL, such that the WHERE-clause may only contain equality conditions combined with AND [3]. In practice, CQs also occur in graph databases, and are known as Basic Graph Patterns in the query language SPARQL [78]. For a more in-depth introduction and study of CQs and related definitions, we refer to Abiteboul, Hull and Vianu [3] and Maier [69].

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

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. For hypergraphs of CQs, we consider them as defined above to be first-order formulas. For such a formula ϕ, the hypergraph H_ϕ corresponding to ϕ is defined as follows: $V(H_\phi) = vars(\phi)$, i.e., the variables occurring in ϕ; and $E(H_\phi) = \{vars(a) \mid a \text{ is an atom in } \phi\}$.

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

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 [63, 69, 73].

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 C of hypergraphs has the *bounded intersection property (BIP)*, if there exists a constant c such that every hypergraph $H \in C$ 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 $V(S) = \bigcup S$.

2.2 Hypergraph Decompositions

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

We are now ready to recall the definitions of hypertree decompositions and generalized hypertree decompositions. We begin, however, by first defining tree decompositions, as they shall play a role in Chapter 5.

A *tree decomposition* [80, 81] (TD) $\mathcal{D}$ of a hypergraph $H = (V(H), E(H))$ is a tuple $\mathcal{D} = \langle T, \chi \rangle$, such that $T = \langle N(T), E(T) \rangle$ is a rooted tree, χ is a node-labelling function with $\chi\colon N(T) \rightarrow 2^{V(H)}$ and the following conditions hold:

(1) for each $e \in E(H)$, there exists a node $u \in N(T)$ with $e \subseteq \chi(u)$;

(2) for each $v \in V(H)$, the set $\{u \in N(T) \mid v \in \chi(u)\}$ is connected in T;

The *width* of a TD $\mathcal{D} = \langle T, \chi, \lambda \rangle$ is the maximum size of the χ-labels over all nodes $u \in T$ minus one, i.e., $width(\mathcal{D}) = \max_{u \in T} |\chi(u)| - 1$. Moreover, the *treewidth* of a hypergraph H, denoted $tw(H)$, is the minimum width over all TDs of H. Condition (2) is called the "connectedness condition". The set $\chi(u)$ is often referred to as the "bag" at node u and we will also call it the "χ-label" of node u.

Next we define hypertree decompositions, which introduce an additional node-labelling function to TDs:

A *hypertree decomposition* [43] (HD) $\mathcal{D}$ of a hypergraph $H = (V(H), E(H))$ is a tuple $\mathcal{D} = \langle T, \chi, \lambda \rangle$, such that $T = \langle N(T), E(T) \rangle$ is a rooted tree, χ and λ are node-labelling functions with $\chi\colon N(T) \rightarrow 2^{V(H)}$ and $\lambda\colon N(T) \rightarrow 2^{E(H)}$ and the following conditions hold:

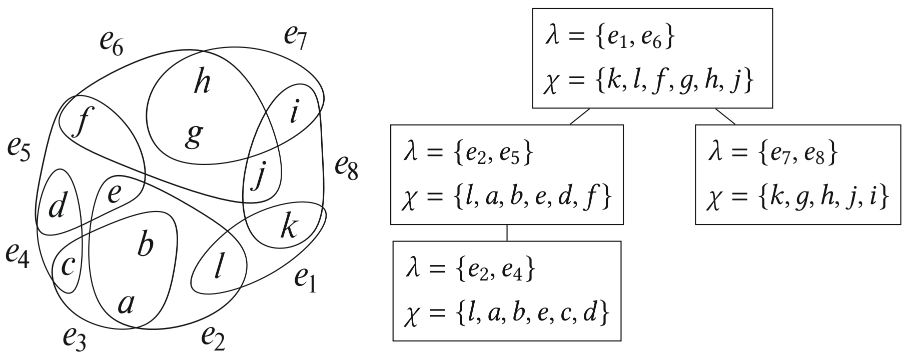

e_6
e_7
e_5
e_8
e_4
e_1
e_3
e_2
h g i f j d e k b c l a
$\lambda = \{e_1, e_6\}$ $\chi = \{k, l, f, g, h, j\}$
$\lambda = \{e_2, e_5\}$ $\chi = \{l, a, b, e, d, f\}$
$\lambda = \{e_7, e_8\}$ $\chi = \{k, g, h, j, i\}$
$\lambda = \{e_2, e_4\}$ $\chi = \{l, a, b, e, c, d\}$

Figure 2.1: An example hypergraph, where the vertices are represented by letters, with explicit edge names, together with a GHD of width 2.

(1) for each $e \in E(H)$, there exists a node $u \in N(T)$ with $e \subseteq \chi(u)$;

(2) for each $v \in V(H)$, the set $\{u \in N(T) \mid v \in \chi(u)\}$ is connected in T;

(3) for each $u \in N(T)$, $\chi(u) \subseteq \bigcup \lambda(u)$;

(4) for each $u \in N(T)$, $\chi(T_u) \cap \left(\bigcup \lambda(u)\right) \subseteq \chi(u)$.

The *width* of an HD $\mathcal{D} = \langle T, \chi, \lambda \rangle$ is the maximum size of the λ-labels over all nodes $u \in T$, i.e., $width(\mathcal{D}) = \max_{u \in T} |\lambda(u)|$. Moreover, the *hypertree width* of a hypergraph H, denoted $hw(H)$, is the minimum width over all HDs of H. As with TDs, Condition (2) is called the "connectedness condition" and condition (4) is referred to as the "special condition" in [43]. The set $\lambda(u)$ will be referred to as either the *edge cover* of u or also the "λ-label" of u.

It has been shown by Gottlob et al. [44] that hypertree width can also be characterised using logical and game-theoretical approaches.

If we drop the special condition from the above definition then we get so-called generalized hypertree decompositions (GHD):

A *generalized hypertree decomposition (GHD)* [43] of a hypergraph $H = (V(H), E(H))$ is a tuple $\langle T, \chi, \lambda \rangle$, where $T = (N(T), E(T))$ is a tree, and χ and λ are node-labelling functions, which map to each node $u \in N$ two sets, $\chi(u) \subseteq V(H)$ and $\lambda(u) \subseteq E(H)$.

The functions χ and λ have to satisfy the following conditions:

(1) for each $e \in E(H)$, there is a node $u \in N(T)$ s.t. $e \subseteq \chi(u)$;

(2) for each $v \in V(H)$, $\{u \in N(T) \mid v \in \chi(u)\}$ is connected in T;

(3) for each $u \in N(T)$, we have that $\chi(u) \subseteq \bigcup(\lambda(u))$.

The second condition is also referred to as the *connectedness condition*. The *width of a GHD* is defined as $\max\{|\lambda(u)| \mid u \in N\}$. The generalized hypertree width (ghw) of a hypergraph is the

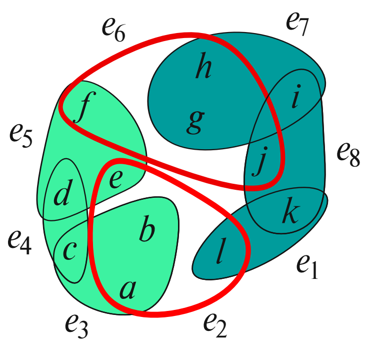


Figure 2.2: Connected components and their respective separator, visually marked.

smallest width of any of its GHDs. Deciding if $ghw(H) \leq k$ for a hypergraph H and fixed k is NP-complete, as one needs to consider exponentially many possible choices for the bag $\chi(u)$ for a given edge cover $\lambda(u)$ [39, 45].

It was shown in [24] 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 2.5.

Example 2.1. An example of a hypergraph is shown in Figure 2.1, 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 [21], 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.

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

2.3 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 $v_1, \ldots, v_h$ and a sequence of edges $e_0, \ldots, e_h$ ($h \geq 1$) 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 $i \in \{1, \ldots, h\}$. 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 2.2. An example for a separator that generates multiple connected components can be seen in Figure 2.2. 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 [4] 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 2.5 below.

2.4 Computing Hypertree Decompositions (HDs)

We briefly recall the basic principles of the det-k-decomp program from [50] for computing Hypertree Decompositions (HDs), which was the first implementation of the original HD algorithm from [43]. 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 $C_i \subseteq C$. 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 $C_1, \ldots, C_\ell$ 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 [43] 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 $n_1, \ldots, n_\ell$, 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, det-k-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.

Example 2.3. We shall demonstrate how a top-down algorithm for computing HDs works by an example run. Formally, we assume our algorithm solves the Check HD problem. In this setting we take as input a hypergraph and have some parameter k and accept and output an HD of our input hypergraph with width k or less, if it exists or reject if no such HD exists for our input hypergraph.
We take as the input hypergraph the one shown in Figure 2.1 and name it H and we fix k to be 2. Let us recall that we are considering an algorithm which works in a top-down manner, first determining a possible root node, then separating the input hypergraph into smaller subhypergraphs and finding hypertree decompositions for those which connect with the root node and proceeding in the same way for the subhypergraphs, until we find an HD of the entire input hypergraph.
For the sake of simplicity, let us assume our algorithm decides on the bag $\chi_r = \{a, b, e, l, f, g, h, j\}$

and the edge cover $\lambda_r = \{e_2, e_6\}$ as the root node. The $[\lambda_r]$-components of H can be seen in Figure 2.2. We get two components, namely $C_1 = \{e_3, e_4, e_5\}$ and $C_2 = \{e_1, e_7, e_8\}$. Thus, our algorithm now tries to find subtrees for each of the two components. Let us look at the case for C_1. Since we are allowed to choose edge covers of size 2, we can simply set $\lambda_{u_1} = \{e_3, e_5\}$ and $\chi_{u_1} = \{a, b, c, d, e, f\}$ and will thus cover the entire component. The case for C_2 is more complex. We need to maintain connectivity, thus the set of vertices $\{h, g, j, l\}$ needs to be covered next, as these vertices form the intersection between the bag of the root and the set of edges that need to be covered (i.e. C_2). If we only restrict ourselves to edges in C_2, we would need to choose all three edges, which we are not allowed to do with k set to 2. The solution is that we can simply reuse e_6 again, and set $\lambda_{u_2} = \{e_1, e_6\}$ and the bag as $\chi_{u_2} = \{l, k, h, g, j\}$. There is only one $[\lambda_{u_2}]$-component of C_2, namely $C_3 = \{e_7, e_8\}$. As this component has only 2 edges, we can trivially cover it by just including all edges and vertices in the edge cover and bag, thus giving us $\lambda_{u_3} = \{e_7, e_8\}$ and $\chi_{u_3} = \{j, g, h, i, k.l\}$.
Thus we are done, having decomposed H and found an HD of width 2.

2.5 Computing GHDs and the Balanced Separator Approach

It was shown in [26] that, even for fixed $k = 2$, deciding if $ghw(H) \leq k$ 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 principle any of these exponentially many bags may be needed to get the desired decomposition

Now let $\lambda(n) = \{e_{i_1}, \ldots, e_{i_\ell}\}$ with $\ell \leq k$. Of course, if we restrict each e_{i_j} to the subedge $e'_{i_j} = e_{i_j} \cap \chi(n)$ and define $\lambda'(n) = \{e'_{i_1}, \ldots, e'_{i_\ell}\}$, then we get $\chi(n) = B(\lambda'(n))$. The key to the tractability shown in [26] 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 [26] for bag-maximal GHDs, that each e'_{i_j} is either equal to e_{i_j} or a subset of e_{i_j} with $|e'_{i_j}| \leq k \cdot b$. Hence, there are only polynomially many choices of subedges e'_{i_j} and also of $\chi(n)$. More precisely, for a given edge e, the set of subedges to consider is defined as follows:

$$f_e(H, k) = \bigcup_{e_1, \ldots, e_j \in (E(H) \setminus \{e\}),\, j \leq k} 2^{(e \cap (e_1 \cup \cdots \cup e_j))}) \tag{2.1}$$

In [24], 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 $\chi(n) = B(\lambda'(n))$.

Algorithm 2.1: `SequentialBalSep` [25]

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

Algorithm 2.2: Function `ComputeSubhypergraphs`

```
1  Function ComputeSubhypergraphs($H'$: hypergraph, $S_p$: set of special edges,
2                                 $B_u$: set of vertices)
3      $H_{ex} \leftarrow E(H' \cup S_p)$    ▷ an "extended" hypergraph with special edges
4      return $[B_u]$-components of $H_{ex}$
```

Algorithm 2.3: Function `InitBalSepIterator`

```
1 Function InitBalSepIterator($H, H'$: hypergraphs, $S_p$: set of special edges, $k$: integer)
2     $E_{\text{rel}} \leftarrow E(H) \cap (S_p \cup E(H'))$          ▷ relevant edges for finding a bal. sep. of $H'$
3     $combinations \leftarrow E^k_{\text{rel}}$          ▷ the set of all $k$ combinations from edges in $E_{\text{rel}}$
4     $balSeps \leftarrow \{s \mid s \in combinations \wedge s$ is a balanced separator of $H'\}$
5     return an iterator that produces one element of $balSeps$ at a time
```

Balanced Separator Approach. Fischl, Gottlob, Longo and Pichler introduce in [24] the Balanced Separator Approach for computing GHDs. It is based on the use of *balanced separators* and extends ideas from [6]. The idea is to use 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 Chapter 3.

Overview of the Sequential Balanced Separator Algorithm [25] We provide here the full pseudo-code of the sequential Balanced Separator algorithm, which can be seen in Algorithm 2.1, since it will prove vital to establish the correctness of our parallel GHD algorithm

We proceed to provide an overview. The algorithm takes as input a hypergraph H and it needs a parameter k. It either returns NULL if no GHD of width k or less can be found, or it returns exactly such a GHD. On lines 1 to 3 we see the main procedure of the algorithm, which simply consists of first making H globally visible (i.e. accessible) and afterwards returns the result of the recursive procedure `Decompose` with input H and the empty set.
The recursive function `Decompose` has thus a primary role in this algorithm. As input it takes a hypergraph H' (not to be confused with the input hypergraph H) and a set of special edges S_p. These are simply edges that act like normal edges, but which appear in the GHD in the edge cover of leaf nodes only. The lines 5 to 6 form a base case of the recursive function, when we have an input with only one edge or special edge. In this case it returns a trivial node with the entire hypergraph and special edge as bag and respectively edge cover. Another base case is seen on lines 7 to 12, when we have in total 2 edges or special edges. In this case, the produced GHD will have exactly two nodes, each with one edge or special edge.
The algorithm next produces an iterator over all balanced separators of H' of size k or less. We can see a pseudo-code for the function `InitBalSepIterator` in Algorithm 2.2. We note here that a given implementation of this function need not materialise all balanced separators at once, but only needs to find one after the other.

On lines 14 to 27, the algorithm iterates over all choices of separators in this iterator. Note that in case there is no balanced separator of size k or less, we skip the while loop and immediately return NULL on line 28. In case we do have some balanced separators, we fix the current choice of edge cover on line 15 and set the bag on line 16 to the union of all edges in λ_u. Next, on lines 18 to 24, we compute first the subhypergraphs for the currently chosen B_u. A detailed pseudo-code for this function is given in Algorithm 2.2. It first unifies the edges and special edges, and then computes the connected components and returns these.
On line 19, Algorithm 2.1, recursively tries to find decompositions for the components. We note here that $c.H$ (resp. $c.S_p$) is understood as the set of normal (resp. special) edges within the component c. One important thing we have to do here is to add $\{B_u\}$ as a new special edge. Special edges need to appear in the leaf nodes of GHDs. This will allow us to "piece together" a complete GHD from the decompositions of the components. If any recursive call returns NULL, then the loop is exited on line 24, and on line 25 we detect this and skip the current choice of balanced separator. This corresponds to backtracking in our search for finding a GHD for the current subhypergraph H'. If no recursive call returns NULL, then we call the function `BuildGHD` and return its output. This function takes as input a bag, an edge cover, and a set of decompositions. It uses the bag and edge cover to form a root, and attaches the decompositions to the root by rerooting the decompositions looking for the special edge which was added on line 19. For more technical details we refer the interested reader to [25]. The here presented sequential balanced separator algorithm will serve as the basis for the parallel GHD algorithm we will develop and present in Chapter 3.

Chapter 3

Parallel Computation of Generalized Hypertree Decompositions

In Section 1.3, we motivated the need for hypergraph decompositions to help improve the ability of existing RDBMSs and CSP solvers. There has been recent progress in providing better practical systems to produce hypergraph decompositions, more specifically for GHDs [25, 85]. However, as we will show in Section 3.5 of this chapter, they fail to produce optimal GHDs for a large number of instances from the HyperBench dataset [25]. In this line of research, we set out to introduce a new decomposition method to provide significant improvements on the existing state of the art.

To this end, we begin by first providing a number of general algorithmic improvements in Section 3.2. These improvements target all combinatorial algorithms and are not limited to the parallel decomposition method which we introduce and test as part of this chapter. The improvements cover preprocessing of the input hypergraph. This preprocessing enables us to reduce the size of the input, thus reducing the runtime needed to find a decomposition. Afterwards, we can apply modifications to the output decomposition, such that it becomes a decomposition of the initial hypergraph, before the preprocessing. We proceed to introduce a series of preprocessing steps, show that they can be applied in a don't-care non-deterministic fashion, and we prove that correctness of these rules.

Next we explore the use of parallelisation in Section 3.3. As we explained as part of Section 1.4, there are a number of challenges that need to be addressed to effectively utilise multi-core systems for the purpose of computing hypergraph decomposition. We first introduce our chosen strategy for parallelising the balanced separator algorithm, and then proceed to explain how our chosen design addresses the challenges we have outlined. We also briefly introduce the idea of using a hybrid method that combines our just introduced parallel algorithm with simpler sequential algorithms, and argue that this hybrid system manages to combine the strengths of both systems, quickly reducing the size of the input hypergraph, without introducing unneeded overhead once small subproblems have been reached.

We conclude this chapter by an experimental evaluation of our decomposition method and other methods from the literature [25, 85]. We use as the basis of this evaluation the HyperBench dataset we mentioned earlier. We proceed to show that an ensemble of all our methods can solve 2924 out of 3648 instances optimally, or around ∼80 %.

This work was created in collaboration with Georg Gottlob and Reinhard Pichler and initial work was first published at the Alberto Mendelzon International Workshop on Foundations of Data Management in 2019 [46]. This was next followed by a publication at the International Joint Conference for Artificial Intelligence (IJCAI) 2020 [47] and ultimately an extended journal publication was published by the journal Constraints in 2022 [49].

3.1 Extended Subhypergraphs and their Balanced Separation

An important idea in the algorithm we will present in this chapter is to split the task of constructing GHDs into subtasks of constructing smaller *parts* of the GHD. We shall call these "GHD-fragments". These GHD-fragments can be reordered, a process we shall introduce later in this chapter, and then stitched together to form a GHD of a given hypergraph. This splitting into GHD-fragments is realised by choosing a node u, and splitting the GHD into various subtrees. We note here that we do not have to keep track of whether these subtrees are "above" or "below" the node u in the final GHD we want to construct. This is made possible by the fact that GHDs are not rooted trees. In order to keep track of how to combine the subtrees later on, we introduce the notion of a *special edge*. A special edge s is a vertex set where $s \subseteq V(H)$, indicating exactly the shared vertices between two GHD-fragments. For any given special edge, the goal will be to find a GHD-fragment which covers the special edge in a leaf node, thus allowing for the corresponding other GHD-fragment to be attached without breaking the connectednesss property.

The decomposition algorithm we will present in Section 3.3 will have as its main procedure the recursive function `Decomp`, which takes as input a subset E' of the edges $E(H)$ and a set of special edges Sp. The goal of `Decomp` is to construct a fragment of a GHD, such that every edge $e \in E'$ is covered by some node u' in the GHD-fragment (i.e., $e \subseteq \chi(u')$) and all special edges are covered by some leaf node of this GHD-fragment. Formally, function `Decomp` deals with *extended subhypergraphs* of H in the following sense.

Definition 3.1 (extended subhypergraph)**.** Let H be a hypergraph. An *extended subhypergraph* of H is a tuple $\langle E', Sp \rangle$ with the following properties:

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

We now extend several crucial definitions introduced in [43] for hypergraphs to extended subhypergraphs.

Definition 3.2 (connectedness, components)**.** Let H be a hypergraph, let $U \subseteq V(H)$ be a set of vertices, and let $H' = \langle E', Sp \rangle$ be an extended subhypergraph of H.

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

Let S be a set of edges and special edges with $U = \bigcup S$. Then we will also use the terms $[S]$-connectedness and $[S]$-components as a short-hand for $[U]$-connectedness and $[U]$-components, respectively. Next we give the definition of generalized hypertree decompositions (GHDs) of extended subhypergraphs.

Definition 3.3 (generalized hypertree decomposition)**.** Let H be a hypergraph and let $H' = \langle E', Sp \rangle$ be an extended subhypergraph of H. A generalized hypertree decomposition (GHD) of H' is a tuple $\langle T, \chi, \lambda \rangle$, such that $T = \langle N(T), E(T) \rangle$ is a rooted tree, χ and λ are node-labelling functions and the following conditions hold:

(1) for each $u \in N(T)$, either
a) $\lambda(u) \subseteq E(H)$ and $\chi(u) \subseteq \bigcup \lambda(u)$ or
b) $\lambda(u) = \{s\}$ for some $s \in Sp$ and $\chi(u) = s$;

(2) each $f \in E' \cup Sp$ is "covered" by some $u \in N(T)$, i.e.:
a) if $f \in E'$, then $f \subseteq \chi(u)$;
b) if $f \in Sp$, then $\lambda(u) = \{f\}$ and, hence, $\chi(u) = f$;

(3) for each $v \in \left(\bigcup E'\right) \cup \left(\bigcup Sp\right)$, the set $\{u \in N(T) \mid v \in \chi(u)\}$ is connected in T;

(4) if $\lambda(u) = \{s\}$ for some $s \in Sp$, then u is a leaf of T;

Clearly, H can also be considered as an extended subhypergraph of itself by taking the tuple $\langle E(H), \emptyset \rangle$. Then the GHDs of the extended subhypergraph $\langle E(H), \emptyset \rangle$ and the GHDs of hypergraph H coincide.

In [43], Definition 5.1, a normal form for HDs (instead of GHDs) was introduced. Below, in Definition 3.5, we will carry the notion of normal form over to GHDs of extended subhypergraphs. To this end, it is convenient to first define the set of (possibly special) edges *covered for the first time* by some node or by some subtree of an GHD.

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

Definition 3.5 (normal form). Let $H' = \langle E', Sp \rangle$ be an extended subhypergraph of some hypergraph H and let $\mathcal{D} = \langle T, \chi, \lambda \rangle$ be an GHD of H'. We say that $\mathcal{D}$ is in *normal form*, if for every node p in T and every child node c of p, the following properties hold:

1. There is exactly one $[\chi(p)]$-component C_p of H' such that $C_p = cov(T_c)$;
2. there exists $f \in C_p$ with $f \subseteq \chi(c)$, where C_p is the $[\chi(p)]$-component fulfilling Condition 1;
3. $\left(\bigcup \lambda(c)\right) \cap \chi(p) \subseteq \chi(c)$.

By the connectedness condition, the following property holds in any GHD: if C' is a $[\chi(p)]$-component of H' with $C' \cap cov(T_c) \neq \emptyset$, then $C' \subseteq cov(T_c)$ must hold. That is, $cov(T_c)$ is the union of *one or several* $[\chi(p)]$-components. Condition 1 of the normal form requires there to be *exactly one* $[\chi(p)]$-component C_p of H' satisfying $C_p \subseteq cov(T_c)$.

Condition 2 intuitively requires that some "progress" must be made by the labelling of node c. Hence, in the first place, at least one vertex from $\bigcup C_p$ not already present in $\chi(p)$ must occur in $\chi(c)$. By the connectedness condition, this is only possible if one edge f from C_p occurs in $\lambda(c)$. Hence, by condition (4) of the definition of GHDs, $f \subseteq \chi(c)$ must hold.

We now carry over two key results from [43], whose proofs can be easily adapted to our setting of extended subhypergraphs and are therefore omitted here.

Theorem 3.6 (cf. [43], Theorem 5.4)**.** *Let H' be an extended subhypergraph of some hypergraph H and let $\mathcal{D}$ be an GHD of H' of width k. Then there exists an GHD $\mathcal{D}'$ of H' in normal form, such that $\mathcal{D}'$ also has width k.*

Lemma 3.7 (cf. [43], Lemma 5.8)**.** *Let H' be an extended subhypergraph of some hypergraph H and let $\mathcal{D} = \langle T, \chi, \lambda \rangle$ be an GHD in normal form of H'. Moreover, let p, c be nodes in T such that p is the parent of c and let $C_c \subseteq C_p$ for some $[\chi(p)]$-component C_p of H'. Then the following equivalence holds: C_c is a $[\chi(c)]$-component of H' if and only if C_c is a $[\lambda(c)]$-component of H'.*

In [25], balanced separators were used to design an algorithm for GHD computation. Below, we formally define balanced separators for our notion of extended subhypergraphs and we show that in a GHD, a balanced separator always exists.

Definition 3.8 (balanced separators)**.** Let H' be an extended subhypergraph of some hypergraph H and let $\mathcal{D} = \langle T, \chi, \lambda \rangle$ be an GHD of H'. A node u of T is a *balanced separator*, if the following holds:

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

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

Lemma 3.9. *Let H' be an extended subhypergraph of some hypergraph H and let $\mathcal{D} = \langle T, \chi, \lambda \rangle$ be an HD of H'. Then there exists a balanced separator in $\mathcal{D}$.*

Proof of Lemma 3.9. We show that, given an arbitrary GHD, we can always find a balanced separator as follows: Initially, we set $u = r$ for the root node r of T and distinguish two cases: if $|cov(T_{u_i})| \leq \frac{|E'|+|Sp|}{2}$ holds for every subtree T_{u_i} rooted at a child node u_i of u, then u is a balanced separator and we are done. Otherwise, there exists a child node u_i of u such that $|cov(T_{u_i})| > \frac{|E'|+|Sp|}{2}$ holds for the subtree T_{u_i} rooted at u_i. Of course, there can exist only one such child node u_i. Moreover, by $cov(T^{\uparrow}_{u_i}) \cap cov(T_{u_i}) = \emptyset$, we have $|cov(T^{\uparrow}_{u_i})| < \frac{|E'|+|Sp|}{2}$. Now set $u = u_i$ and repeat the case distinction: if $|cov(T_{u_i})| \leq \frac{|E'|+|Sp|}{2}$ holds for every subtree T_{u_i} rooted at a child node u_i of u, then u is a balanced separator and we are done. Otherwise, there exists a child node u_i of u such that $|cov(T_{u_i})| > \frac{|E'|+|Sp|}{2}$ holds for the subtree T_{u_i} rooted at u_i. Again, there can only be one such u_i. So we set $u = u_i$ and iterate the same considerations. This process is guaranteed to terminate since, eventually, we will reach a leaf node of T. □

3.2 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 chapter. 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.

3.2.1 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 [51, 95] 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 [55], hinge decompositions were introduced to help split CSPs into smaller subproblems. In [34], 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 $C = \{C_1, \dots, C_\ell\}$ with $\ell \geq 2$ denote the $[e]$-components of H. Then we may split H into hypergraphs $H_1 = (V(H_1), E(H_1))$,..., $H_\ell = (V(H_\ell), E(H_\ell))$ 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 $\chi(n) = \bigcup E(H)$.

We shall note that the main reason we consider the trivial Rule 5 is purely technical, in that it allows us to prove that the arbitrary application of all rules, including Rule 5, always leads to a unique normal form. The inclusion of Rule 5 does not do any harm as any practical decomposition method will at any rate check if the currently investigated hypergraph already has edge cover number of k or less.

In Theorems 3.10 and 3.11 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 3.10. *Preprocessing an input hypergraph H via Rules 1 – 5 is correct. 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 $ghw(H) \leq k$ if and only if, for every $i \in \{1, \dots, m\}$, $ghw(H_i) \leq k$ 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 $H_1, \dots, H_\ell$ 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 correctness 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.

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 $C_1, \dots, C_\ell$. 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 $n_1 + \cdots + n_\ell \leq n - 1$ 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 $C_1, \dots, C_\ell$ and let $|C_i| = n_i$. Then H is split into ℓ hypergraphs $H_1, \dots, H_\ell$ 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 $H_1, \dots, H_\ell$ together, the total number of applications of Rule 4 is therefore $\leq (2n_1 + \cdots + 2n_\ell) - \ell$. Together with the inequalities $n_1 + \cdots + n_\ell \leq n - 1$ 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.

Correctness. For the correctness of our reduction system, we have to prove the correctness 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 $\{H_1, \ldots, H_m\}$, 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 $ghw(H) \leq k$ if and only if $ghw(H') \leq k$. 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 $H_1, \ldots, H_\ell$ is the result of a single application of Rule 4 to H. Then $ghw(H) \leq k$ if and only if, for every $i \in \{1, \ldots, \ell\}$, $ghw(H_i) \leq k$ holds. Moreover, in the positive case, a GHD of H of width $\leq k$ can be constructed from GHDs of $H_1, \ldots, H_\ell$ of width $\leq k$ in polynomial time.

Note that we have omitted Rule 5 in this claim, since both the correctness 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 3.2.3. □

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

Proof. Recall that, in Theorem 3.10, 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* [8]. 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 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 [8], this property is succinctly presented as follows:

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

Table 3.1: Overview of the complexity of the four methods considered for ordering hyperedges.

Method of edge ordering	Runtime Worst Case Complexity
Maximal cardinality search ordering	$O(\lvert E(H)\rvert^2)$
Maximal separator ordering	$O(\lvert E(H)\rvert \cdot \lvert V(H)\rvert^3)$
Vertex degree ordering	$O(\lvert E(H)\rvert^2)$
Edge degree ordering	$O(\lvert E(H)\rvert^2)$

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

This case disctinction is rather tedious (especially the cases where Rule 4 is involved) but not difficult. We thus defer the details to Section 3.2.4. □

3.2.2 Finding Balanced Separators Fast

It has already been observed in [50] 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 [50] 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 [27, 92] as a subroutine, this leads to a fairly high complexity – see Table 3.1 – 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 $\lvert\{f : e \cap f \neq \emptyset\}\rvert$, i.e., the number of edges e has a non-empty intersection width.

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 [24] 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 2.5). 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 $(e_1, \ldots, e_k)$, 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 2.5, we get the following set of edge combinations when factoring in all the relevant subedges:

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

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 3.3. But first we present the missing parts of the proofs of Theorems 3.10 and 3.11 in Sections 3.2.3 and 3.2.4, respectively.

3.2.3 Completion of the Proof of Theorem 3.10

It remains to prove Claim A from the proof in Section 3.2.1.

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.

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 $v_2 \notin e_i$.

$\Rightarrow$: 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:

- 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 $C_1, \dots, C_\ell$ 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 [43], 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 $f \neq e$ 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 $n_1, \dots, n_j$ 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 $\lambda(n)$.

3.2.4 Completion of the Proof of Theorem 3.11

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 $H_1, \ldots, H_\ell$ 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 $H_1, \ldots, H_\ell$ 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 $H_1, \dots, H_\ell$ 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 $H_1', \dots, H_\ell'$ 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 $C = \{C_1, \ldots, C_\ell\}$. Or we may first apply Rule 4 to H to get the hypergraphs $H_1, \ldots, H_\ell$ 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 $C = \{C_1, \ldots, C_\ell\}$ 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 $H_1, \ldots, H_\ell$ with $E(H_i) = C_i \cup \{e\}$ for every $i \in \{1, \ldots, \ell\}$. 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 $H_1, \ldots, H_\ell$ as before.

Case 1.2. If $e_1 \nsubseteq e$, then also $e_2 \nsubseteq e$ 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 $H_1, \ldots, H_{i-1}, H'_i, H_{i+1}, \ldots H_\ell$ 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 $C' = \{C_1, \dots, C_{i-1}, C'_i, C_{i+1}, \dots, C_\ell\}$ 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 $H'_1, \dots, H'_\ell$ 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 $D_\ell, \dots, D_m$ with $m > \ell$, then we may apply Rule 4 to H_ℓ and replace H_ℓ by $H'_\ell, \dots, H'_m$ 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 $H_1, \dots, H_\ell$ 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, $e_1 \cap \big(V(D_1) \cup \dots \cup V(D_{m-1}) \cup e_2\big) = \emptyset$. 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 $G_1, \dots, G_\ell$ 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 $S_1, \dots, S_\alpha$ 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 $T_1, \dots, T_\beta$ 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 $T_1, \dots, T_\beta$ 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 $S_1, \dots, S_\alpha$ with $\alpha \geq 1$ denote the $[e_1]$-components of H inside R_n and let $T_1, \dots, T_\beta$ 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 the hypergraphs F'_i with $i \le n-1$ 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 $i \le n-1$. In other words, the vertices in $e_1 \setminus d$ only occur in a single edge of F'_i with $i \le n-1$, namely in the edge e_1. We may therefore apply Rule 1 multiple times to each of the hypergraphs F'_i with $i \le n-1$. 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 $i \le n-1$.

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. □

3.3 Parallelisation Strategy

As described in more detail below, we use a divide and conquer method, based on the Balanced Separator Approach [25], which we recalled in Section 2.5. 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

Algorithm 3.1: Parallel Balanced Separator algorithm

```
Type: Comp=($H$: Graph, $Sp$: 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($H'$: Graph, $Sp$: 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          $sep$ := FindBalSep($H'$, $Sp$, $M$)
9          if $sep = \emptyset$ then
10             break
11         subSep := $sep$
12         repeat                                                     ▷ SupEdgeLoop
13             comps := ComputeSubhypergraphs($H'$, $Sp$,$subSep$)     ▷ returns Comp set
14             ch := a channel
15             for $c \in comps$ do
16                 go ch ← Decomp($c.H$, $c.Sp \cup \bigcup subSep$)
17             while any recursive call is still running do
18                 out ← ch:                                          ▷ waits on channel
19                 if $out$ = Reject then
20                     subSep = NextSubedgeSep($subSep$)
21                     continue SubEdgeLoop
22                 return Accept                              ▷ found decomposition
23         until $subSep = \emptyset$
24     until $sep = \emptyset$
25     return Reject                                       ▷ exhausted search space
```

width k exists. For the computation we use the modern programming language Go [18], which has a model of concurrency based on [57].

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 3.1, line 16. They communicate over *channels*. Using a channel *ch* is indicated by ← *ch* for *receiving* from a channel, and by *ch* ← for *sending* to *ch*.

3.3.1 Overview

Algorithm 3.1 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 on line 2 of Algorithm 3.1. The procedure `Decomp` takes two arguments, a subhypergraph H' of H for the current subcomponent considered, and a set Sp of special edges. Together, these form an extended subhypergraph, as introduced in Section 4.1. The special edges indicate the balanced separators encountered so far, as can be seen on 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 thus to find a GHD $\mathcal{D}$ of $\langle H', Sp \rangle$ that fulfils the conditions given in Definition 3.3.

Hence, the central procedure `Decomp` in Algorithm 3.1, when initially called on line 2, checks if there exists a GHD of the desired width of the extended subhypergraph $(H, \emptyset)$, that is, a GHD of hypergraph H itself.

The recursive procedure `Decomp` has its base case on 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 iterates over all subedge variants of any balanced separator. New balanced separators are produced with the subprocedure `FindBalSep`, used on line 8 of Algorithm 3.1, and detailed in Function FindBalSep. After a separator is found, `Decomp` computes the new subcomponents on line 13. Then goroutines are started using recursive calls of `Decomp` for all found subcomponents. If any of these calls returns Reject, seen on 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 on 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.

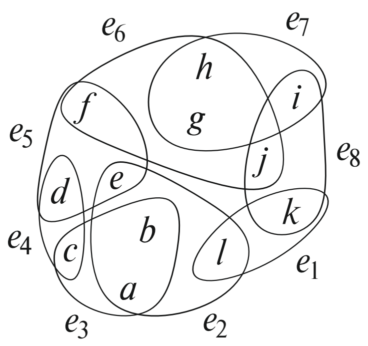


Figure 3.1: An example hypergraph, where the vertices are represented by letters, with explicit edge names.

3.3.2 Parallel Search for Balanced Separators

Before describing our implementation, we define some needed notions. For the search for balanced separators within an extended subhypergraph $\langle H', Sp\rangle$, 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 $\{1, \ldots, |E^*|\}$ according to this ordering. We can then define the following notion.

Definition 3.12. A k*-combination* for an ordered set of edges E^* is a k-tuple of integers $(x_1, \ldots, x_k)$, where $1 \leq x_i \leq |E^*|$ and $x_1 < \cdots < x_k$. 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 3.1. Consider the hypergraph H from Figure 3.1. 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 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 on 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:

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 ← 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 ∈ NextSeparator($M_i$) do
11         if IsBalanced(sep, H', Sp) then
12             ch ← sep                                       ▷ send sep to FindBalSep
13     return                                                 ▷ no separator found within $M_i$
```

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 Figure 3.2.

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 on 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 3.1 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,

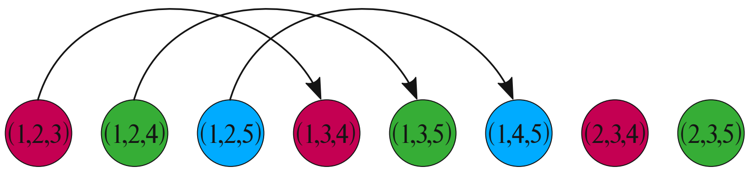


Figure 3.2: Using k-combinations to split the workspace. Shown here with 3 workers, and $k = 3$ and $|E^*| = 5$.

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 on line 6 of Algorithm 3.1, *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.

3.3.3 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 3.1, on 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 on 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 3.1 - 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 [50]. 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 [15].

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 3.5.

3.3.4 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 3.13. *The algorithm for checking the* ghw *of a hypergraph given in Algorithm 3.1 is sound and complete. More specifically, Algorithm 3.1 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 [25]. Let us refer to it as `SequentialBalSep`. A detailed proof of the soundness and completeness of `SequentialBalSep` is given in [25]. For convenience of the reader, we recall the pseudo-code description of `SequentialBalSep` from [25] in the Preliminaries. 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 [24], 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 3.3.2, 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 $\langle H', Sp \rangle$ 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 3.2.2: 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 3.2.2, 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 initialise `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. □

3.3.5 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 3.2. 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 3.2 and combines the Balanced Separator algorithm from Sections 3.3.1–3.3.3 and det-k-decomp recalled in Section 2.4.

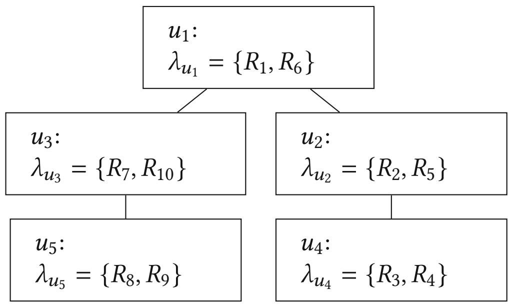
This combination is motivated by two observations: The Balanced Separator algorithm is very effective at splitting large hypergraphs into smaller ones and in negative cases, where it can quickly stop the computation if no balanced separator for a given subcomponent exists. It is slower for smaller instances 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.

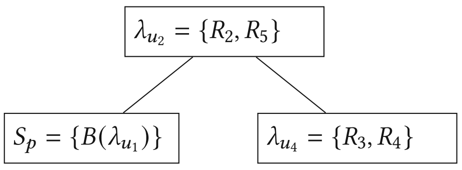
3.4 An Illustrative Example

In order to illustrate how the parallel Balanced Separator algorithm we proposed in Section 3.3 works, we will consider an example run of the algorithm. The pseudo-code can be seen in



u_1:
$\lambda_{u_1} = \{R_1, R_6\}$
u_3:
$\lambda_{u_3} = \{R_7, R_{10}\}$
u_2:
$\lambda_{u_2} = \{R_2, R_5\}$
u_5:
$\lambda_{u_5} = \{R_8, R_9\}$
u_4:
$\lambda_{u_4} = \{R_3, R_4\}$

(a) GHD $\mathcal{D}$ of hypergraph H from Section 3.4



$\lambda_{u_2} = \{R_2, R_5\}$
$S_p = \{B(\lambda_{u_1})\}$
$\lambda_{u_4} = \{R_3, R_4\}$

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

Figure 3.3: Visualisations of the GHD constructed as part of Section 3.4 and the GHD-fragments used for its construction.

Algorithm 3.1. We consider as the input the hypergraph $H = (V, E)$ with $V = \{x_1, \ldots, x_{10}\}$ and

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

In other words, H is a essentially a cycle of size 10. A generalized hypertree decomposition $\mathcal{D}$ of H is shown in Figure 3.3a. Please note that for the sake of simplicity, we omit the χ-labels here. The reason for this is that in our parallel Balanced Separator algorithm, we simply take the union of all vertices of the λ-label to form the bags. Thus we can also choose a simpler representation of GHDs which only concerns itself with the λ-labels. We note here that this example avoids the need to consider subedges. The reason for this is the fact that the algorithm first finds a balanced separator (a set of edges), and only in case of encountering a failure case does it begin to iterate over all possible subedges of edges in this balanced separator, as long as such a combination is still a balanced separator itself. This means subeges are only computed locally when needed. Since our particular example never encounters a failure case, we also do not need to consider subedges.

We now walk through the parallel Balanced Separator algorithm. We assume that we run the algorithm with hypergraph H and parameter $k = 2$.

In the SeparatorLoop on line 7 to line 24, the algorithm searches for a λ-label, until it finds a successful one. By "successful" we mean that the correct execution of the function `Decomp` returns an Accept on line 22. To keep things simple in our discussion below, we will directly choose a successful one with the understanding that this particular λ-label will eventually be selected by the program unless another successful one has already been found before.

Main program. The main program simply calls the function `Decomp` with the input of H and the empty set, as we have no special edges initially. Below we will simply enumerate the recursive calls to `Decomp` in a hierarchical fashion, where for example the Call $x.y.z$ signifies the zth recursive call within Call $x.y$. Call $x.y$ is then the yth recursive call of function `Decomp` from within Call x. And finally Call x is just the xth call from the Main program. Note that x will always be just 1 here, since the main program of Algorithm 3.1 only calls `Decomp` exactly once.

Call 1 of function `Decomp` with parameters $H'.E = \{R_1, \ldots, R_{10}\}$, $H'.Sp = \emptyset$.

Since the condition of the base case is not satisfied, the SeparatorLoop will be entered. It will eventually try $sep = \{R_1, R_6\}$. We shall already think of this as the λ-label of node u_1, which will form the root of the GHD of H. This splits H' into 2 components $c_1 = \{R_2, R_3, R_4, R_5\}$ and $c_2 = \{R_7, R_8, R_9, R_{10}\}$, which are computed on line 13. The algorithm now proceeds to call function `Decomp` with the input c_1 (resp. c_2) and the set of special edges being $\{B(\lambda_{u_1})\}$.

As we shall work out next, Call 1.1 of function `Decomp` for the component c_1 will return true based on the GHD-fragment $\mathcal{D}_{1.1}$ shown in Figure 3.3b. Likewise, Call 1.2 of function `Decomp` for the component c_2 will return true based on the GHD-fragment $\mathcal{D}_{1.2}$, which we do not visualise here as its structurally analogous to $\mathcal{D}_{1.1}$

The left leaf node of $\mathcal{D}_{1.1}$ contains the special edge $\{B(\lambda_{u_1})\}$, which acts as a placeholder for the node u_1 with label $\lambda_{u_1} = \{R_1, R_6\}$ from the current call of `Decomp`.

The GHD of the successful Call 1 of function `Decomp` is then obtained by taking GHD-fragment $\mathcal{D}_{1.1}$, rerooting it to the leaf node with λ-label $\{B(\lambda_{u_1})\}$, then replacing it by the node u_1 with $\lambda_{u_1} = \{R_1, R_6\}$. We proceed analogously for the GHD-fragment $\mathcal{D}_{1.2}$, but attach it in the end to the same node u_1. The final GHD $\mathcal{D}$ (shown in Figure 3.3a) is thus obtained . We do not visualise the rerooting process here, as it has already been described in Section 2.5.

Call 1.1 of function `Decomp` with parameters $H'.E = \{R_2, R_3, R_4, R_5\}$, $H'.Sp = \{B(\lambda_{u_1})\}$.

On line 8, we will eventually choose $\lambda_{u_2} = \{R_2, R_5\}$. This choice leads to two components being produced on line 13, namely $c_3 = \{R_3, R_5\}$ and c_4, which only consists of the special edge $\{B(\lambda_{u_1})\}$. Thus, `Decomp` is called the first time – as Call 1.1.1 – with input $\emptyset$ for the set of edges, and the set $\{B(\lambda_{u_1}), B(\lambda_{u_2})\}$ as the set of special edges. And the second time the function `Decomp` is called – as Call 1.1.2 – with the input $\{R_3, R_5\}$ and the set $\{B(\lambda_{u_2})\}$ as the set of special edges. The GHD-fragment associated with Call 1.1 can be seen in Figure 3.3b. This is produced by combining the outputs from Calls 1.1.1 and 1.1.2 via rerooting to the leaf node with λ-label of $\{B(\lambda_{u_2})\}$ and replacing and attaching it to $u2$.

Call 1.1.1 of function `Decomp` with parameters $H'.E = \emptyset$ and $H'.Sp = \{B(\lambda_{u_1}), B(\lambda_{u_2})\}$. This call immediately returns true since we have reached the base case on lines 4 - 5. The corresponding GHD-fragment $\mathcal{D}_{1.1.1}$ consists of two nodes, one with λ-label $\{B(\lambda_{u_1})\}$ and the other with λ-label $\{B(\lambda_{u_2})\}$.

Call 1.1.2 of function `Decomp` with parameters $H'.E = \{R_3, R_5\}$, $H'.Sp = \{B(\lambda_{u_2})\}$. On line 8, we assume that $\lambda_{u_4} = \{R_3, R_4\}$ will be chosen. This leads to a single component with no regular edges, and special edges $\{B(\lambda_{u_2}), B(\lambda_{u_5})\}$. This leads to another recursive call, which we will directly cover here since it falls into a base case. As with Call 1.1.1, this function call of `Decomp` with input $\emptyset$ for $c.H$ and special edges $\{B(\lambda_{u_2}), B(\lambda_{u_5})\}$ will produce a GHD-fragment with two nodes. After rerooting the output of this recursive call, the GHD-fragment associated with our current Call 1.1.2 will consist of two nodes: u_4, with λ-label λ_{u_4} as defined above, and a leaf node with λ-label of $\{B(\lambda_{u_2})\}$.

Call 1.2 of function `Decomp` with parameters $H'.E = \{R_7, R_8, R_9, R_{10}\}$, $H'.Sp = \{B(\lambda_{u_1})\}$. The execution of this function call is very similar to the calls discussed above. Below, we therefore do not discuss in detail the remaining recursive calls inside Call 1.2. Instead, we only list for each such call the parameters, the balanced separator, and the corresponding GHD-fragments.

In Call 1.2 of function `Decomp`, eventually the balanced separator with $\lambda_{u_3} = \{R_7, R_{10}\}$ will be chosen, which gives rise to the recursive Calls 1.2.1 and 1.2.2 (both on line 16) of function `Decomp`, which we briefly discuss below.

Call 1.2.1 of function `Decomp` with parameters $H'.E = \{R_8, R_9\}$, $H'.Sp = \{B(\lambda_{u_3})\}$. As in the Call 1.1.2, we now have an input consisting of two edges and a single special edge. Analogously to Call 1.1.2, also Call 1.2.1 returns true and the corresponding GHD-fragment $\mathcal{D}_{1.2.1}$ consists of 2 nodes: the root node with λ-label $\{R_8, R_9\}$ and its child nodes with λ-label $\{B(\lambda_{u_3})\}$.

Call 1.2.2 of function `Decomp` with parameters $H'.E = \emptyset$ and $H'.Sp = \{B(\lambda_{u_1}), B(\lambda_{u_3})\}$. Again, we have an input consisting of no regular edges and two special edges. Analogously to the Call 1.1.1, also Call 1.2.2 returns true and the corresponding GHD-fragment $\mathcal{D}_{1.2.2}$ consists of 2 nodes: the root node with λ-label λ_{u_1} and its child node with λ-label $\{B(\lambda_{u_3})\}$.

We can now construct the GHD-fragment $\mathcal{D}_{1.2}$ of the successful Call 1.2 by taking GHD-fragment $\mathcal{D}_{1.2.2}$, rerooting it to the leaf node with λ-label $\{B(\lambda_{u_3})\}$, and then replacing it by the node u_3 with $\lambda_{u_3} = \{R_7, R_{10}\}$ from Call 1.2 and appending the GHD-fragment $\mathcal{D}_{1.2.1}$ – after rerooting it to the leaf node with λ-label $\{B(\lambda_{u_3})\}$ and removing it – below this node u_3.

3.5 Experimental Evaluation and Results

We have performed our experiments on the HyperBench benchmark from [24] 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 experiments [48], in addition to the source code of our Go programs[1] are publicly available. Together with the benchmark instances, which are detailed below and also publicly available, this ensures the reproducibility of our experiments.

3.5.1 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 [24] 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

[1]See: `https://github.com/cem-okulmus/BalancedGo`

Table 3.2: Overview of the instances from HyperBench and their average sizes by group, as well as sizes of groups themselves.

Instances					
Group	Avg sizes		Arity		Size
	$\lvert V\rvert$	$\lvert E\rvert$	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

still undetermined. In this extended evaluation, we performed the evaluation on a larger set of instances when compared with the original paper [47], to reflect the newest version of the benchmark, published in [24]. We provide a more detailed overview of the various instances, grouped by their origin, in Table 3.2. 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 [24], thus ensuring the direct comparability of our test results. To enforce these limits and run the experiments, we used the *HTCondor* software [88], 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.

3.5.2 Empirical Results

The key results from our experiments are summarised in Table 3.4, with Table 3.3 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 3.3, namely the results of the Hybrid approach (from

Table 3.3: Overview of previous results: number of instances solved and running times (in seconds) for producing optimal-width GHDs in [24] and [85]

Instances	Decomposition Methods							
Group	NewDetKDecomp by [24]				HtdSMT by [85]			
	#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 3.4: 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 *ensemble*			
	#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

Section 3.3.5) is separately shown in a section of the table. As a reference point, we considered on one hand the *NewDetKDecomp* library from [24] and also the SAT Modulo Theory based solver *HtdSMT* from [85]. 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 3.3 and 3.4 are hypergraphs of (resp.) CSP or CQ instances from real world applications.

In the left part of Table 3.4, we report on the following results obtained with our Hybrid Approach described in Section 3.3.5, 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

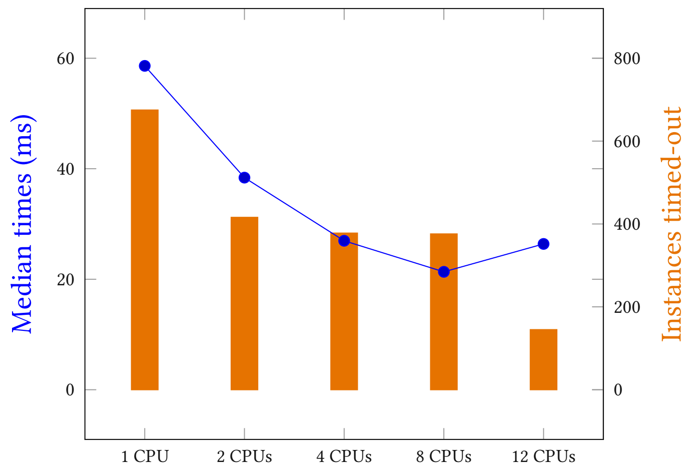


Figure 3.4: Study of the performance gain w.r.t. the number of CPUs used

Table 3.5: Comparison of BalancedGo, HtdSMT and TULongo on the PACE 2019 Challenge, Track 2a. Columns t_{avg} and t_{sum} show the average time and the total time, respectively, over all private instances.

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 [68]	70	38	105.58	1.11

Table 3.6: Overview of exclusively solved instances of HyperBench for each decomposition method.

Method	#exclusively solved
BalancedGo *ensemble*	**284**
NewDetKDecomp	11
HtdSMT	67

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. [24] are also such an “ensemble”, combining the results of three different GHD algorithms presented in [24]. 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

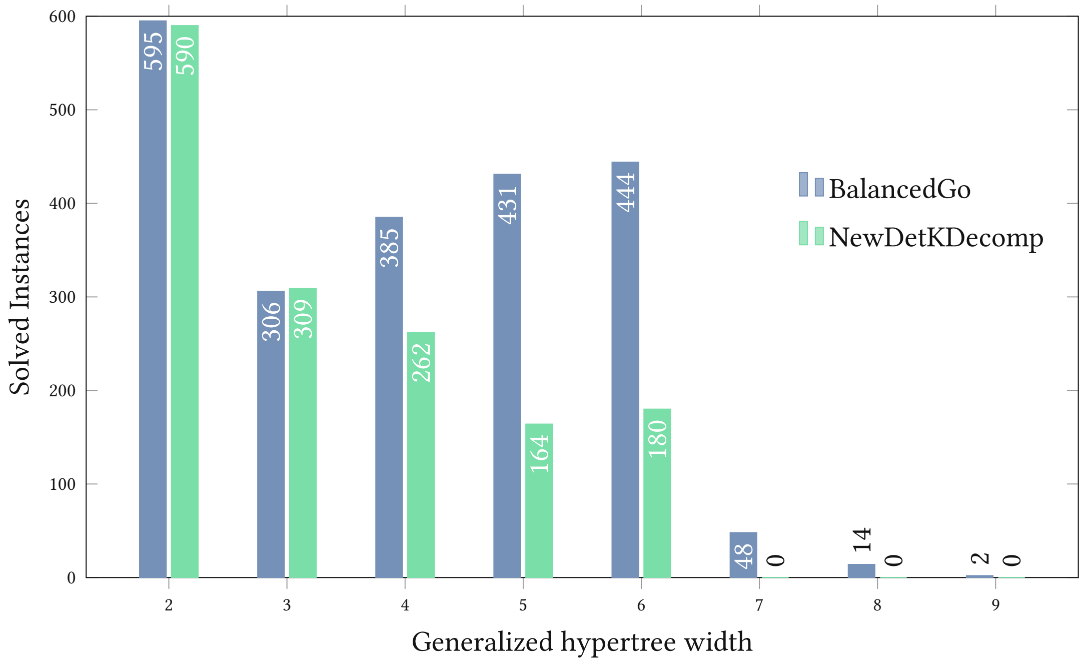


Figure 3.5: Overview of the distribution of the *ghw* of the solved instances.

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 3.4 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 timeouts. 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 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.

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 3.6. We see that while our approach clearly dominates this metric, there are still many cases where other methods were more effective.

In Figure 3.5 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* ≥ 3, our result signifies an increase of over 70 % in solved instances when compared with [24]. 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 78% 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 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 3.4 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 [24], which, as mentioned, combines the results of three different GHD algorithms and also beating the total for HtdSMT [85].

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 [20], 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 [68] and reran them again using the same time and memory constraints. The results can be seen in Table 3.5. *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.

3.6 Summary

In this chapter, we have presented several generally applicable algorithmic improvements for hypergraph decomposition algorithms, which together reduce the overall search space needed for finding GHDs, without affecting the correctness of any decomposition methods which integrate them. The rules for simplifying hypergraphs also permit a don't-care non-determinism in how they are applied, leading to a unique normal form of the reduced hypergraph.

We then proceeded to present a novel parallel algorithm for computing GHDs, which is based on the sequential Balanced Separator Approach from Fischl, Gottlob, Longo and Pichler [25]. We also show the practical applicability of our design by providing a publicly available implementation, called BalancedGo. This implementation was written in the programming language Go and uses the concept of goroutines to reduce the need for explicit synchronisation as much as possible. We show its practical applicability via a detailed experimental evaluation using the benchmark dataset HyperBench and comparing our method against the state of the art in computing GHDs. We demonstrated that our method BalancedGo provided significant improvements, determining the exact *ghw* of 78% of all hypergraphs of HyperBench.

Chapter 4

Novel Parallel Algorithm for Hypertree Decomposition

In Chapter 3, we presented a parallel algorithm for computing GHDs. Our next goal is trying to use the concept of balanced separators for computing HDs. The most obvious reason is the complexity of the CheckHD problem, which is computable in polynomial time, whereas solving the analogous CheckGHD is NP-hard, even for $k \geq 2$ [39]. The second reason is from an observation from Fischl, Gottlob, Longo and Pichler [25]. For all the hypergraphs where they could compute both their hypertree width and generalized hypertree width, they found that they were the same. This suggests that there may be many instances in the real world where it might be a better idea to just look for HDs. The use of balanced separators for HDs faces a number of challenges, however, as we have outlined in Section 1.4.

This chapter begins with Section 4.1 which introduces a new form of extended hypergraph – different from the one in Chapter 3 – , which we will need for technical reasons and will serve as the underlying object that our algorithm will take as input and work with. For this new type of hypergraph, we present a number of lemmas and theorems, introducing a new concept of hypertree decomposition with restrictions to how it can connect, up and down, to other decompositions. These will prove crucial to then prove the correctness of our algorithm.

We next introduce our actual algorithm itself in Section 4.2, provide the pseudo-code description of it in Algorithm 4.1 and this is followed by a proof of its correctness in Section 4.3. We follow this by a detailed example run of our algorithm in Section 4.4.

In order to truly strive for a competitive algorithm, we next introduce a number of optimisations to the base algorithm given earlier. These improvements to our algorithm can be found in Section 4.5, where we also state the full optimised definition of log-k-decomp in Algorithm 4.2.

We end this chapter by providing an experimental evaluation in Section 4.6. We compare our method with the state of the art in computing HDs. As the benchmark we use the HyperBench dataset, already featured in Chapter 3. For the interested reader, we provide in Section 4.7 additional experiments and further details on our implementation, instead of the more high-level experiments of Section 4.6.

This work was created in collaboration with Georg Gottlob, Matthias Lanzinger and Reinhard Pichler. Our work was published at the Symposion on Principles of Database Systems (PODS) 2022 [37]. The journal ACM Transactions on Database Systems (TODS) also invited us to publish an updated version of this work as one of only four "best of PODS 2022" papers.

4.1 Connection Subhypergraphs and their Balanced Separation

As in Section 3.1 in the previous Chapter 3, we again introduce another type of extended hypergraph, and then proceed to define the needed definitions of components, hypertree decomposition and balanced separation on this new type of extended subhypergraph.

The key idea of our algorithm is to split the task of constructing an HD into subtasks of constructing *parts* of the HD, which will be referred to as “HD-fragments” in the sequel. These HD-fragments can later be stitched together to form an HD of a given hypergraph. This splitting into HD-fragments is realised by choosing a node u of the HD and splitting the HD into one subtree above node u and possibly several subtrees below u. In order to keep track of how to combine these subtrees later on, we introduce the notion of *special edges*. Intuitively, a special edge is the set $\chi(u)$ of vertices for some node u in the HD, and it is used to keep track of the interface between the HD-fragment “above” node u (we will denote this part of the HD as $T_u^{\uparrow}$) and the HD-fragments at subtrees below node u. Conversely, for each of the subtrees T_{u_i} rooted at the child nodes u_i of u, we have to keep track of the interface to $\chi(u)$ in the form of a set *Conn* of vertices, which is the intersection $\chi(T_{u_i}) \cap \chi(u)$.

At the heart of our decomposition algorithm in Section 4.2 will be a recursive function `Decomp`, which takes as input a subset E' of the edges $E(H)$, a set of special edges Sp, and a set of vertices $Conn$. The goal of `Decomp` is to construct a fragment of an HD, such that every edge $e \in E'$ is covered by some node u' in the HD-fragment (i.e., $e \subseteq \chi(u')$), all special edges are covered by some leaf node of this HD-fragment (hence, these are the interfaces to the HD-fragments “below”) and $Conn$ must be fully contained in $\chi(r)$ of the root r of this HD-fragment (hence, this is the interface to the HD-fragment “above”). Formally, function `Decomp` deals with *extended subhypergraphs* of H in the following sense. We shall note that this is a different concept than the one introduced in Chapter 3.

Definition 4.1 (connection subhypergraph)**.** Let H be a hypergraph. An extended subhypergraph with connection interfaces or short *connection subhypergraph* of H is a triple $\langle E', Sp, Conn\rangle$ with the following properties:

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

As we said, we shall now proceed to extend the notions defined over *extended subhypergraphs* in Section 3.1 to connection subhypergraphs. Note that we will talk about HDs here, instead of GHDs. We shall also introduce a different type of normal form, one that will deviate in one crucial point from the one introduced in [43]. Aside from these differences, the definitions are kept analogous between the two chapters.

Definition 4.2 (connectedness, components)**.** Let H be a hypergraph, let $U \subseteq V(H)$ be a set of vertices, and let $H' = \langle E', Sp, Conn\rangle$ be a connection subhypergraph of H.

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

Let S be a set of edges and special edges with $U = \bigcup S$. Then we will also use the terms $[S]$-connectedness and $[S]$-components as a short-hand for $[U]$-connectedness and $[U]$-components, respectively. Observe that the set *Conn* plays no role in the above definition of connectedness and components. This is in contrast to our definition of hypertree decompositions (HDs) of connection subhypergraphs, which we give next.

Definition 4.3 (hypertree decomposition)**.** Let H be a hypergraph and let $H' = \langle E', Sp, Conn\rangle$ be a connection subhypergraph of H. A hypertree decomposition (HD) of H' is a tuple $\langle T, \chi, \lambda\rangle$, such that $T = \langle N(T), E(T)\rangle$ is a rooted tree, χ and λ are node-labelling functions and the following conditions hold:

(1) for each $u \in N(T)$, either
a) $\lambda(u) \subseteq E(H)$ and $\chi(u) \subseteq \bigcup \lambda(u)$ or
b) $\lambda(u) = \{s\}$ for some $s \in Sp$ and $\chi(u) = s$;

(2) each $f \in E' \cup Sp$ is "covered" by some $u \in N(T)$, i.e.:
a) if $f \in E'$, then $f \subseteq \chi(u)$;
b) if $f \in Sp$, then $\lambda(u) = \{f\}$ and, hence, $\chi(u) = f$;

(3) for each $v \in \left(\bigcup E'\right) \cup \left(\bigcup Sp\right)$, the set $\{u \in N(T) \mid v \in \chi(u)\}$ is connected in T;

(4) for each $u \in N(T)$, $\chi(T_u) \cap \left(\bigcup \lambda(u)\right) \subseteq \chi(u)$;

(5) if $\lambda(u) = \{s\}$ for some $s \in Sp$, then u is a leaf of T;

(6) the root r of T satisfies $Conn \subseteq \chi(r)$.

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

We also want to define a notion of normal form over HDs of connection subhypergraphs. In order to do this, we first need to define the set of (possibly special) edges *covered for the first time* by some node or by some subtree of an HD.

Definition 4.4. Let $H' = \langle E', Sp, Conn \rangle$ be a connection subhypergraph of some hypergraph H and let $\mathcal{D} = \langle T, \chi, \lambda \rangle$ be an HD of H'. For a node $u \in T$, we write $cov(u)$ to denote the set of edges and special edges *covered for the first time* at u, i.e.: $cov(u) = \{f \in E' \cup Sp \mid f \subseteq \chi(u)$ and for all ancestor nodes u' of u, $f \not\subseteq \chi(u')$ holds$\}$. For a subtree T' of T, we define $cov(T') = \bigcup_{u \in T'} cov(u)$.

Definition 4.5 (normal form of connection hypergraphs)**.** Let $H' = \langle E', Sp, Conn \rangle$ be a connection subhypergraph of some hypergraph H and let $\mathcal{D} = \langle T, \chi, \lambda \rangle$ be an HD of H'. We say that $\mathcal{D}$ is in *normal form*, if for every node p in T and every child node c of p, the following properties hold:

1. There is exactly one $[\chi(p)]$-component C_p of H' such that $C_p = cov(T_c)$;
2. there exists $f \in C_p$ with $f \subseteq \chi(c)$, where C_p is the $[\chi(p)]$-component satisfying Condition 1;
3. $\chi(c) = \left(\bigcup \lambda(c)\right) \cap \left(\bigcup C_p\right)$, where again C_p is the $[\chi(p)]$-component satisfying Condition 1.

Condition 3 is the only place where we deviate from the normal form in [43]. The purpose of Condition 3 in [43] is to make sure that $\chi(c)$ is uniquely determined whenever $\lambda(c)$, $\chi(p)$, and the $[\chi(p)]$-component C_p from Condition 1 are known. However, there also would have been other choices to achieve this goal. Our Condition 3 chooses $\chi(c)$ *minimally*. That is, to ensure the special condition, $\chi(c)$ must contain all vertices from $\bigcup \lambda(c)$ that occur in $\chi(T_c)$. Since all edges in C_p are covered at some node in T_c, all vertices from $\left(\bigcup \lambda(c)\right) \cap \left(\bigcup C_p\right)$ must occur in $\chi(c)$. On the other hand, there is no need to add further vertices to $\chi(c)$, since vertices not occurring in $\bigcup cov(T_c)$ can never violate the connectedness condition at node c as long as we stick to our strategy of choosing $\chi(u)$ minimally also for all nodes $u \in T_c$. In contrast, Condition 3 in [43] chooses $\chi(c)$ *maximally*. That is, also all vertices in $\left(\bigcup \lambda(c)\right)$ that occur in $\chi(p)$ are added to $\chi(c)$. This deviation from the normal form in [43] is crucial since, in our construction of an HD, we will be able to derive the possible sets C_p as soon as we have $\lambda(p)$ and $\lambda(c)$ but we will "know" $\chi(p)$ only much later in the algorithm.

As in Section 3.1, we now carry over two key results from [43].

Theorem 4.6 (cf. [43], Theorem 5.4)**.** *Let H' be a connection subhypergraph of some hypergraph H and let $\mathcal{D}$ be an HD of H' of width k. Then there exists an HD $\mathcal{D}'$ of H' in normal form, such that $\mathcal{D}'$ also has width k.*

Lemma 4.7 (cf. [43], Lemma 5.8)**.** *Let H' be a connection subhypergraph of some hypergraph H and let $\mathcal{D} = \langle T, \chi, \lambda \rangle$ be an HD in normal form of H'. Moreover, let p, c be nodes in T such that p is the parent of c and let $C_c \subseteq C_p$ for some $[\chi(p)]$-component C_p of H'. Then the following equivalence holds: C_c is a $[\chi(c)]$-component of H' if and only if C_c is a $[\lambda(c)]$-component of H'.*

Note that our deviation from [43] in the definition of the χ-label of nodes in a normal-form HD is inessential, since the "downward" components in an HD are not affected by adding or removing vertices from the parent node to the χ-label of the child node. However, for our purposes, we need a slightly stronger version of the above lemma: recall that the HD construction in [43] proceeds in a strict top-down fashion. Hence, when dealing with $\lambda(c)$, the bag $\chi(p)$ is already known. This is due to the fact that, initially at the root r, we have $\chi(r) = \bigcup \lambda(r)$ by the special condition. And then, whenever $\lambda(c)$ is determined and $\chi(p)$ plus a $[\chi(p)]$-component are already known, also $\chi(c)$ can be computed. However, in our HD algorithm, which "jumps into the middle" of the HD to be constructed, we only have $\lambda(p)$ (but not $\chi(p)$) available when determining $\lambda(c)$. Hence, we need to slightly extend the above lemma to the following corollary, which follows from Lemma 4.7 by an easy induction argument over the distance from the root of the HD:

Corollary 4.8. *Let H' be a connection subhypergraph of some hypergraph H and let $\mathcal{D} = \langle T, \chi, \lambda \rangle$ be an HD in normal form of H'. Moreover, let p, c be nodes in T such that p is the parent of c and let $C_c \subseteq C_p$ for some $[\lambda(p)]$-component C_p of H'. Then the following equivalence hods: C_c is a $[\chi(c)]$-component of H' if and only if C_c is a $[\lambda(c)]$-component of H'.*

As the algorithm we will present relies on them, we also extend the definition of balanced separators to connection subhypergraphs.

Definition 4.9 (balanced separators)**.** Let H' be a connection subhypergraph of some hypergraph H and let $\mathcal{D} = \langle T, \chi, \lambda \rangle$ be an HD of H'. A node u of T is a *balanced separator*, if the following holds:

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

Lemma 4.10. *Let H' be a connection subhypergraph of some hypergraph H and let $\mathcal{D} = \langle T, \chi, \lambda \rangle$ be an HD of H'. Then there exists a balanced separator in $\mathcal{D}$.*

We will skip a separate proof of Lemma 4.10, as the exact same arguments in the proof of Lemma 3.9 carry over to connection subhypergraphs and thus serve as a proof of Lemma 4.10 as well.

4.2 The `log`-k-`decomp` Algorithm

We now describe the main ideas of algorithm `log`-k-`decomp`. A pseudo-code description of `log`-k-`decomp` is shown in Algorithm 4.1.

Algorithm `log`-k-`decomp` aims at constructing an HD in normal form according to Definition 4.5 of width $\leq k$ for a given hypergraph H and integer $k \geq 1$. The task of constructing an HD is split into subtasks that can then be processed in parallel. At the heart of `log`-k-`decomp` is the recursive function `Decomp`: it takes as input a connection subhypergraph H' of H in the form of parameter H' of H (with two fields $H'.E$ and $H'.Sp$ for the sets of edges and special edges of H', respectively) plus parameter *Conn* for the interface of the HD-fragment to be constructed

Algorithm 4.1: `log-k-decomp`

```
Type: Comp=($E$: Edge set, $Sp$: Special Edge set)
Input: $H$: Hypergraph
Parameter: $k$: width parameter
Output: true if $hw$ of $H \leq k$, else false
1  begin
2      $H_{comp}$ := Comp($E$: $H$, $Sp$: $\emptyset$)
3      foreach $\lambda_r \subseteq H$ s.t. $1 \leq |\lambda_r| \leq k$ do                ▷ RootLoop
4          $comps_r$ := $[\lambda_r]$-components of $H_{comp}$
5          foreach $y \in comps_r$ do
6              $Conn_y$ := $V(y) \cap \bigcup \lambda_r$
7              if not(Decomp($y$, $Conn_y$)) then
8                  continue RootLoop                                   ▷ reject this root
9          return true
10     return false                                                    ▷ exhausted search space
11 function Decomp($H'$: Comp, $Conn$: Vertex set)
12     if $|H'.E| \leq k$ and $|H'.Sp| = 0$ then                         ▷ Base Cases
13         return true
14     else if $|H'.E| = 0$ and $|H'.Sp| = 1$ then
15         return true
16     foreach $\lambda_p \subseteq H$ s.t. $1 \leq |\lambda_p| \leq k$ do               ▷ ParentLoop
17         $comps_p$ := $[\lambda_p]$-components of $H'$
18         if $\exists i$ s.t. $|comps_p[i]| > \frac{|H'|}{2}$ then
19             $comp_{down}$ := $comps_p[i]$                                ▷ found child comp.
20         else
21             continue ParentLoop
22         if $V(comp_{down}) \cap Conn \not\subseteq \bigcup \lambda_p$ then
23             continue ParentLoop                                     ▷ connect. check
24         foreach $\lambda_c \subseteq H$ s.t. $1 \leq |\lambda_c| \leq k$ do             ▷ ChildLoop
25             $\chi_c$ := $\bigcup \lambda_c \cap V(comp_{down})$
26             if $V(comp_{down}) \cap \bigcup \lambda_p \not\subseteq \chi_c$ then
27                 continue ChildLoop                                  ▷ connect. check
28             $comps_c$ := $[\chi_c]$-components of $comp_{down}$
29             if $\exists i$ s.t. $|comps_c[i]| > \frac{|H'|}{2}$ then
30                 continue ChildLoop
31             foreach $x \in comps_c$ do
32                 $Conn_x$ := $V(x) \cap \chi_c$
33                 if not(Decomp($x$, $Conn_x$)) then
34                     continue ChildLoop                              ▷ reject child
35             $comp_{up}$ := $H' \setminus comp_{down}$                           ▷ pointwise diff.
36             $comp_{up}.Sp = comp_{up}.Sp \cup \{\chi_c\}$
37             if not(Decomp($comp_{up}$, $Conn$)) then
38                 continue ChildLoop                                  ▷ reject child
39             return true                                             ▷ $hw$ of $H' \leq k$
40     return false                                                    ▷ exhausted search space
```

with the parts "above" in the final HD. It returns "true" if an HD-fragment of width $\leq k$ of H' exists and "false" otherwise. The top-level calls to function `Decomp` (line 7) are from the main program of `log-`k`-decomp` which, in a loop (lines 3 – 9), searches for the λ-label of the root node r of the desired HD of H. By the special condition, we have $\chi(r) = \bigcup \lambda(r)$. Hence, the $[\lambda(r)]$-components (computed on line 4) coincide with the $[\chi(r)]$-components. Function `Decomp` is called (on line 7) for each of the connection subhypergraphs of H corresponding to the $[\lambda(r)]$-components.

The base case of function `Decomp` is reached (lines 12 – 15) when the existence of such an HD-fragment is trivial, i.e.: either there are at most k edges and no special edges left; or there is no edge and only one special edge left. In these cases, the desired HD-fragment simply consists of a single node whose λ-label either consists of the $\leq k$ edges or of the single special edge, respectively.

Function `Decomp` is controlled by two nested loops (lines 16 – 39 for the outer loop and lines 24 – 39 for the inner loop), which search for the λ-labels of two adjacent nodes p and c of the desired HD-fragment, such that p is the parent and c is the child. The idea of determining two nodes p and c is that, in an HD, we can determine $\chi(c)$ from $\lambda(c)$ if we know $\lambda(p)$ and the $[\lambda(p)]$-component covered by the subtree T_c rooted at c, see Corollary 4.8 and Definition 4.5.

We want node c to be a balanced separator of the connection subhypergraph H'. By Lemma 4.10, a balanced separator is guaranteed to exist. To find a balanced separator c, we have to make sure that node c satisfies the two conditions of Definition 4.9, i.e.: (1) all of the subtrees rooted at a child of c cover at most half of the edges and special edges in H' and (2) the subtree $T_c^{\uparrow}$ "above" c covers strictly fewer than half of the edges and special edges in H'. For the second condition, observe that $comp_{down}$ (chosen on line 19) is meant to be covered precisely by T_c. Note that, w.l.o.g., we are searching for an HD in normal form. This is why we may assume that T_c covers exactly one $[\lambda(p)]$-component, namely $comp_{down}$. Further observe that the edges and special edges covered by $T_c^{\uparrow}$ and the set $comp_{down}$ partition the edges and special edges in H'. Hence, checking if $comp_{down}$ contains more than half of H' (on line 18) is equivalent to checking condition (2), i.e., $T_c^{\uparrow}$ covers strictly fewer than half of the edges and special edges in H'. In order to check that c also satisfies the first condition of Definition 4.9, we have to compute all $[\lambda(c)]$-components inside $comp_{down}$ (line 28) and check that the size of each of them is at most half of the size of H' (line 29). Again, since we are only interested in HDs in normal form, we may assume here that each subtree rooted at a child of c covers exactly one of these $[\lambda(c)]$-components.

If such a balanced separator $\lambda(c)$ together with the λ-label $\lambda(p)$ at its parent node has been found, several checks have to be performed to make sure that the HD-fragment under construction satisfies the connectedness condition. For instance, all vertices in the intersection of $Conn$ (i.e., the interface of the HD-fragment currently being constructed with the remaining HD "above" this HD-fragment) with component C_p (i.e., a component "below" node p) also have to occur in $\bigcup \lambda(p)$ (line 22).

Suppose that all these checks succeed. From $\lambda(p)$ and $\lambda(c)$, we can compute $\chi(c)$ according to Condition 3 of the normal form introduced in Definition 4.5 (line 25). In the HD $\mathcal{D}'$ to be constructed for the connection subhypergraph H', the edges and special edges of H' can be split into 3 disjoint categories:

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

The edges and special edges covered by $\chi(c)$ are done and need no further consideration. The edges and special edges in the second and third category are taken care of by recursive calls to the function `Decomp` (lines 33 and 37). To this end, we compute all $[\chi(c)]$-components $C_1, \ldots, C_m$ (line 28). Now suppose that $C_1, \ldots, C_\ell$ with $1 \leq \ell \leq m$ are the $[\chi(c)]$-components inside the $[\lambda(p)]$component C_p. Then the function `Decomp` is called recursively for each of the $[\chi(c)]$-components $C_1, \ldots C_\ell$ (line 33). In the call for component C_i, the interface $Conn_i$ is obtained simply as the intersection of the vertices in C_i and in $\chi(c)$ (line 32). All of the remaining $[\chi(c)]$-components are taken care of by the HD-fragment “above” c, which we try to construct in another recursive call of function `Decomp` (line 37). In this recursive call, $\chi(c)$ is added as yet another special edge – in addition to the edges and special edges in the $[\chi(c)]$-components outside C_p. The additional special edge in the recursive call for the HD-part “above” node c and the interfaces $Conn$ defined for each of the components as the intersection against $\chi(c)$, in the recursive calls for the HD-parts “below” node c ensure that we can (provided that all recursive calls of function `Decomp` are successful) stitch together the HD-fragments of these recursive calls to an HD-fragment of the connection subhypergraph H' of H.

To sum up, if all recursive calls return “true” then the overall result of this call to function `Decomp` is successful and returns “true” (line 39) . If at least one of the recursive calls returns “false”, then we have to search for a different label $\lambda(c)$ (in the next iteration of the “ChildLoop“). If eventually all candidates for $\lambda(c)$ have been tried out and none of them was successful, then we have to search for a different label $\lambda(p)$ of the parent node p (in the next iteration of the “ParentLoop“) and restart the search for $\lambda(c)$ from scratch. Only when also all candidates for $\lambda(p)$ have been tried out and none of them was successful, then function `Decomp` returns the overall result “false” (line 40).

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

Theorem 4.11. *Algorithm `log-k-decomp` correctly checks for given hypergraph H and integer $k \geq 1$, if $hw(H) \leq k$ holds. The algorithm is realised by a main program and the recursive function `Decomp`, whose recursion depth is bounded logarithmically in the number of edges of H, i.e., $O(\log(|H|)$.*

Proof. The size of the connection subhypergraphs in the calls of function `Decomp` in the main program can only be bounded by the size of H itself. However, in every subsequent execution of `Decomp` for some connection subhypergraph $(H'.E, H'.Sp, Conn)$, we always choose node c as a balanced separator. By Lemma 4.10, such a balanced separator always exists. The connection subhypergraphs in the recursive calls are therefore guaranteed to have size at most $\lceil \frac{|H'|}{2} \rceil$. Note that the rounding up is necessary because the new special edge $\chi(c)$ is added in the recursive call for the HD-fragment above c. Without this special edge, this component is guaranteed to be strictly smaller than $\frac{|H'|}{2}$. At any rate, also with the upper bound $\lceil \frac{|H'|}{2} \rceil$ on the size of the connection subhypergraphs of H in the recursive calls and with the additional calls of Decomp from the main program, we thus get an upper bound $O(\log(|H|)$ on the recursion depth. □

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

We emphasise two further important properties of algorithm log-k-decomp: First, it should be noted that the logarithmic bound on the recursion depth does not restrict the form of the HD in any way. In particular, it does not imply a logarithmic bound on the depth of the HD. The bound on the recursion depth is achieved by our novel approach of constructing the HD by recursively "jumping" to a balanced separator of the HD-fragment to be constructed rather than constructing the HD in a strict top-down manner as proposed in previous approaches [43, 50].

Second, we stress that it is crucial in our approach that we search for appropriate λ-labels for *a pair* (p, c) *of nodes*, where p is the parent of c. The rationale is that we need the λ-label of the parent in order to determine $\chi(c)$ from $\lambda(c)$. And only when we know $\chi(c)$, we can be sure, which edges are indeed covered by $\chi(c)$. This knowledge is crucial to guarantee that all of the recursive calls of function Decomp have to deal with anconnection subhypergraph whose size is halved, which in turn guarantees the logarithmic upper bound on the recursion depth. This strategy is significantly different from all previous approaches of decomposition algorithms. In [25], a parallel algorithm for generalised hypertree decompositions is presented. There, the problem of determining the χ-label of the balanced separator is solved by adding a big number of subedges to the hypergraph so that one may assume that $\chi(u) = \bigcup \lambda(u)$ holds for every node u. Clearly, this addition of subedges, in general, leads to a substantial increase of the hypergraph. In [6], a preliminary attempt to parallelise the computation of HDs was made without handling pairs of nodes. However, in the absence of $\lambda(p)$, we cannot determine $\chi(c)$ from $\lambda(c)$. Consequently, we do not know which edges covered by $\bigcup \lambda(c)$ are ultimately covered by $\chi(c)$. Hence, all the edges covered by $\bigcup \lambda(c)$ would have to be added to the recursive call of Decomp for the HD-part "above" c, thus destroying the balancedness and the logarithmic upper bound on the recursion depth.

By Theorem 4.11, Algorithm log-k-decomp guarantees a logarithmic bound on the recursion depth and thus provides a good basis for a parallel implementation. Nevertheless it still leaves room for several improvements. For instance, we can define also negative base cases to detect the overall answer "false" faster, we can restrict the edges that may possibly be used in the λ-labels of a connection subhypergraph (and provide them as an additional parameter of the function Decomp), etc. These ideas and several further improvements – together with the pseudo-code of the resulting improved algorithm – are presented in Section 4.5.

4.3 Correctness Proof of `log-`k`-decomp`

We prove the soundness and completeness of the algorithm `log-`k`-decomp` given in Algorithm 4.1 separately. The polynomial-time upper bound on the construction of an HD in case of a successful run of the algorithm (i.e., if it returns "true") will be part of the soundness proof.

It is convenient to first prove the following claim:

Claim A. *In every call of function* `Decomp` *in Algorithm 4.1 with parameters* $(H', Conn)$ *it is guaranteed that* $Conn \subseteq V(H')$ *holds with* $V(H') = \big(\bigcup H'.E\big) \cup \big(\bigcup H'.Sp\big)$.

Proof of Claim A. The proof is by induction on the call depth of the recursive function `Decomp`.

induction begin. The top-level calls of function `Decomp` on line 7 are with parameters $(y, Conn_y)$, where $Conn_y$ is defined on line 6 as $Conn_y = V(y) \cap \bigcup \lambda(r)$. Hence, $Conn_y \subseteq V(y)$ clearly holds.

induction step. Suppose that Claim A holds for every call of function `Decomp` down to some call level n and suppose that function `Decomp` is called recursively during execution of `Decomp` at call level n. Suppose that this execution of `Decomp` is with parameters $(H', Conn)$. The only places where function `Decomp` is called recursively are lines 33 and 37. More specifically, `Decomp` is called with parameters $(x, Conn_x)$ on line 33 and with parameters $(comp_{up}, Conn)$ on line 37. We have to show that both $Conn_x \subseteq V(x)$ (on line 33) and $Conn \subseteq V(comp_{up})$ (on line 37) hold. On line 33, the condition is trivially fulfilled, since $Conn_x$ is defined on line 32 as $Conn_x = V(x) \cap \chi(c)$.

It remains to consider the call of function `Decomp` on line 37. Suppose that $comps_c$ on line 28 is of the form $comps_c = \{x_1, \ldots, x_\ell\}$. By the definition of components in Definition 4.2, H' (that is, $H'.E \cup H'.Sp$) can be partitioned into the following disjoint subsets:

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

We thus have $V(H') = \bigcup_{i=1}^{\ell} V(x_i) \cup V(y) \cup V(z)$ with $V(y) \subseteq \chi(c)$. By construction (line 28), all components x_i are contained in $comp_{down}$. Hence, we actually have $V(H') = V(comp_{down}) \cup \chi(c) \cup V(z)$. The recursive call of function `Decomp` on line 37 is with the edges and special edges in z plus $\chi(c)$ as an additional special edge. Hence, we have $V(comp_{up}) = V(z) \cup \chi(c)$ when `Decomp` is called on line 37 with parameters $(comp_{up}, Conn)$. It is therefore sufficient to show that $Conn \subseteq V(z) \cup \chi(c)$ holds.

By the induction hypothesis, we may assume that $Conn \subseteq V(H')$ holds. The check on line 22 ensures that $Conn \subseteq \bigcup \lambda(p)$ holds for some edge set $\lambda(p)$. Moreover, the check on line 26 ensures that $\big(\bigcup \lambda(p)\big) \cap V(comp_{down}) \subseteq \chi(c)$. In total, we thus have $Conn \cap V(comp_{down}) \subseteq \chi(c)$. Together with $V(H') = V(comp_{down}) \cup \chi(c) \cup V(z)$ and $Conn \subseteq V(H')$, we may thus conclude

$Conn \subseteq V(z) \cup \chi(c)$ and, therefore, $Conn \subseteq V(comp_{up})$ (on line 37). Hence, also the call of function `Decomp` on line 37 satisfies Claim A. □

Soundness Proof. Suppose that algorithm `log-`k`-decomp` returns "true". That is, for some value of $\lambda(r)$, each call of function `Decomp` on line 7 returns "true". We have to show that there exists an HD of width $\leq k$ of H. To construct such an HD, we take $\lambda(r)$ as the λ-label of the root r of this HD. By the special condition of HDs, we must take $\chi(r) = \bigcup \lambda(r)$. Hence, the $[\chi(r)]$-components and $[\lambda(r)]$-components of H coincide and $comps_r$ computed on line 4 contains all $[\chi(r)]$-components of H.

Now suppose that function `Decomp` is sound (we will prove the correctness of this assumption below), i.e., for an arbitrary connection subhypergraph $(E', Sp, Conn)$ of H, if function `Decomp` returns "true" on input $((E', Sp), Conn)$, then there exists an HD of width $\leq k$ of $(E', Sp, Conn)$. Hence, if the calls of function `Decomp` on line 7 all yield "true", then, by assuming the soundness of `Decomp`, we may conclude that an HD of width $\leq k$ exists for each connection subhypergraph of H of the form $(y.E, y.Sp, Conn_y)$. Let each of these HDs be denoted by $\mathcal{D}[y]$ with tree structure $T[y]$ and let $r[y]$ denote the root of $T[y]$. Then we can construct an HD of H by taking r with $\lambda(r)$ from line 3 and $\chi(r) = \bigcup \lambda(r)$ as root node and appending the HD-fragments $\mathcal{D}[y]$ to r, such that the root nodes $r[y]$ of the trees $T[y]$ become child nodes of r. It is easy to verify that the resulting decomposition is an HD of H, and this HD can be constructed in polynomial time from the HD-fragments $\mathcal{D}[y]$. It remains to show that function `Decomp` is sound.

Soundness of function `Decomp`. For an arbitrary connection subhypergraph $(E', Sp, Conn)$ of H, let function `Decomp` return "true" on input $((E', Sp), Conn)$; we have to show that then there exists an HD of width $\leq k$ of $(E', Sp, Conn)$. Moreover, we have to show that by materialising the decompositions implicitly constructed in the recursive calls of function `Decomp`, an HD of width $\leq k$ of $(E', Sp, Conn)$ can be constructed in polynomial time whenever `Decomp` returns "true". The proof is by induction on $|E'| + |Sp|$.

induction begin. Suppose that $|E'| + |Sp| = 1$ and that function `Decomp` returns "true". Hence, we either have $|E'| = 1$ and $|Sp| = 0$ or we have $|E'| = 0$ and $|Sp| = 1$. In either case, an HD of this connection subhypergraph can be obtained with a single node u by setting $\lambda(u) = \{f\}$ and $\chi(u) = f$, where f is the only (special) edge in $E' \cup Sp$. This decomposition clearly satisfies all conditions of an HD according to Definition 4.3, the only non-trivial part being Condition (6): we have to verify $Conn \subseteq \chi(u)$. By Claim A above, we know that in every call of function `Decomp`, $Conn$ is a subset of the vertices in $E' \cup Sp$. Now in case $|E'| + |Sp| = 1$ holds, we have $E' \cup Sp = \{f\}$ for a single (special) edge f and, therefore, $\chi(u) = f = \left(\bigcup E'\right) \cup \left(\bigcup Sp\right)$. Hence, we indeed have $Conn \subseteq \chi(u)$.

induction step. Now suppose that $|E'| + |Sp| > 1$ and that function `Decomp` returns "true". This means that one of the return-statements on lines 13, 15, or 39 is executed. Actually, line 15 can be excluded for $|E'| + |Sp| > 1$. Now consider the remaining two lines 13 and 39. If the return-statement on line 13 is executed, then we have $|E'| \leq k$ and $|Sp| = 0$. In this case, analogously to the induction begin, the desired HD consists of a single node u with $\lambda(u) = E'$

and $\chi(u) = \bigcup E'$. Again, all conditions of an HD according to Definition 4.3 are easy to verify; in particular, the proof argument for Condition (6) is the same as above.

It remains to consider the case that "true" is returned on line 39. This means that, for a particular value of $\lambda(p)$ (chosen on line 16) and of $\lambda(c)$ (chosen on line 24), all recursive calls of function `Decomp` (on lines 33 and 37) return "true". By the induction hypothesis, we may assume that for each of the connection subhypergraphs processed by these recursive calls of `Decomp`, an HD of width $\leq k$ exists. Note that we are making use of Claim A here in that we may assume that all recursive calls of `Decomp` are with properly defined connection subhypergraphs (in particular, the vertex set supplied as second parameter is covered by the edges and special edges in the first parameter of each such call). Now look at these recursive calls: we are studying a call of function `Decomp` with parameters H' and $Conn$, where H' consists of a set $H'.E$ of edges and a set $H'.Sp$ of special edges. That is, function `Decomp` is processing the connection subhypergraph $(H'.E, H'.Sp, Conn)$. The current call of function `Decomp` apparently has chosen labels $\lambda(p)$ and $\lambda(c)$ for nodes p and c, such that all checks on lines 18, 22, 26, and 29 are successful in the sense that program execution continues with these values of $\lambda(p)$ and $\lambda(c)$. In particular, there exists a $[\lambda(p)]$-component $comp_{down}$ of $(H'.E, H'.Sp, Conn)$, satisfying the conditions $V(comp_{down}) \cap Conn \subseteq \bigcup \lambda(p)$ (line 22) and $V(comp_{down}) \cap \bigcup \lambda(p) \subseteq \chi(c)$ (line 26).

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

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

From the first two kinds of sets of edges and special edges, the following connection subhypergraphs are constructed, for which function `Decomp` is then called recursively on lines 33 and 37:

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

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

- First take the HD of $H^{\uparrow}$. We shall refer to this HD as $\mathcal{D}^{\uparrow}$. Let r denote the root node of $\mathcal{D}^{\uparrow}$. By $Conn^{\uparrow} = Conn$, we have $Conn \subseteq \chi(r)$.

- Recall that $\chi(c)$ was added as a special edge to the connection subhypergraph $H^{\uparrow}$. Hence, by Definition 4.3, the HD $\mathcal{D}^{\uparrow}$ has a leaf node u with $\lambda(u) = \{\chi(c)\}$ and $\chi(u) = \chi(c)$. Now we replace node u in $\mathcal{D}^{\uparrow}$ by node c with $\lambda(c)$ and $\chi(c)$ according to the current execution of function `Decomp`. Moreover, for every $f \in H'.Sp$ with $f \subseteq \chi(c)$, we append a fresh child node c_f to c with $\lambda(c_f) = \{f\}$ and $\chi(c_f) = f$. It is easy to verify that the resulting decomposition (let us call it $\mathcal{D}'$) is an HD of the connection subhypergraph that contains all edges and special edges of H' except for the ones in any of the x_i's. In particular, node r with $Conn \subseteq \chi(r)$ is still the root of HD $\mathcal{D}'$.

- Now we take the HDs $\mathcal{D}_i$ of the connection subhypergraphs $(x_i.E, x_i.Sp, Conn_i)$ and append them as subtrees below c in $\mathcal{D}'$, i.e.: the root nodes of the HDs $\mathcal{D}_i$ become child nodes of c. Let us refer to the resulting decomposition as $\mathcal{D}$. It remains to show that $\mathcal{D}$ indeed is an HD of width $\leq k$ of the connection subhypergraph $(H'.E, H'.Sp, Conn)$ of H. The width is clear, since all HD-fragments of $\mathcal{D}$ and also $\lambda(c)$ have width $\leq k$. It is also easy to verify that every edge in $H'.E$ is covered by some node in $\mathcal{D}$ and every special edge in $H'.Sp$ is covered by some leaf node in $\mathcal{D}$. Moreover, also the connectedness condition holds inside each HD-fragment (by the induction hypothesis) and between the various HD-fragments. The latter condition is ensured by the definition of components in Definition 4.2 and by the fact that any two connection subhypergraphs processed by the various recursive calls of function `Decomp` can only share vertices from $\chi(c)$.

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

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

- $E' = cov(T')$;
- let B denote those nodes in T which are outside T' but whose parent node is in T'; then $Sp = \{\chi(u) \mid u \in B\}$.
- the root r' of T' is different from the root of T; hence, r' has a parent node p in T;
- $Conn = V(H') \cap \bigcup \lambda(p)$.

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

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

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

Completeness Proof. Suppose that hypergraph H has an HD of width $\leq k$. We have to show that then algorithm `log-`k`-decomp` returns "true". By Theorem 4.6, H also has an HD $\mathcal{D} = \langle T, \chi, \lambda \rangle$ of width $\leq k$ in normal form. Note that, in order to apply Theorem 4.6, we are considering H as a connection subhypergraph $(E(H), \emptyset, \emptyset)$ of itself. Let r denote the root of T. If algorithm `log-`k`-decomp` has not already returned "true" before, it will eventually try $\lambda(r)$ in the foreach-statement on line 3. Let $C_1, \ldots, C_\ell$ with $C_i \subseteq E(H)$ for each i denote the $[\lambda(r)]$-components (and hence also the $[\chi(r)]$-components) of H. By the normal form of $\mathcal{D}$, we know that r has ℓ child nodes $u_1, \ldots, u_\ell$ with $cov(T_{u_i}) = C_i$. Now let $Conn_i = V(C_i) \cap \bigcup \lambda(r)$ for $i \in \{1, \ldots, \ell\}$. Hence, $(C_i, \emptyset, Conn_i)$ is a connection subhypergraph of H. Moreover, by the connectedness condition, we have $Conn_i \subseteq \chi(u_i)$. Hence, HD $\mathcal{D}$ restricted to the subtree T_{u_i} is in fact an HD of width $\leq k$ of the connection subhypergraph $(C_i, \emptyset, Conn_i)$.

Now suppose that function `Decomp` is complete on induced subhypergraphs of H (we will prove the correctness of this assumption below). By this we mean that if $(E', Sp, Conn)$ is an induced subhypergraph of H, then function `Decomp` returns "true" on input $(E', Sp), Conn)$. Hence, the calls of function `Decomp` on line 7 all yield "true". Therefore, program execution exits the foreach-loop and executes the return-statement on line 9. That is, algorithm `log-`k`-decomp` returns "true" as desired. It remains to show that function `Decomp` is complete on induced subhypergraphs.

Completeness of function `Decomp`. Consider an arbitrary induced subhypergraph $(E', Sp, Conn)$ of H. We have to show that then function `Decomp` returns "true" on input $((E', Sp), Conn)$. We proceed by induction on $|E'| + |Sp|$.

induction begin. Suppose that $|E'| + |Sp| = 1$. That is, we either have $|E'| = 1$ and $|Sp| = 0$ or we have $|E'| = 0$ and $|Sp| = 1$. In the first case, "true' is returned via the statement on line 13; in the second case, "true' is returned via the statement on line 15.

induction step. Now suppose that $|E'| + |Sp| > 1$. If $|E'| \leq k$ and $|Sp| = 0$, then the return-statement on line 13 is executed and the function returns "true". It remains to consider the case that $|E'| > k$ or $|Sp| > 1$ holds. By Lemma 4.10, the HD $\mathcal{D}'$ induced by H' has a balanced separator. Let us refer to this balanced separator as the node c in $\mathcal{D}'$. By the balancedness, it can be easily verified that c must satisfy $\lambda(c) \subseteq E(H)$ (that is, c is not a leaf node with $\lambda(c) = \{f\}$ for some special edge f). We distinguish two cases:

Case 1. Suppose that c is the root node of $\mathcal{D}'$. Recall that in our definition of induced subhypergraphs, the root of the corresponding tree T' is different from the root r of $\mathcal{D}$. Hence, c has a parent node in $\mathcal{D}$. Let us refer to this parent node as p. If function `Decomp` has not already returned "true" before, it will eventually try $\lambda(p)$ in the foreach-statement on line 16. Due to the normal form of $\mathcal{D}$, all of H' is a single $[\lambda(p)]$-component. Hence, the if-condition on line 18 is satisfied and $comp_{down}$ is assigned all of H' on line 19. The connectedness check on line 22 succeeds, since p is the parent of the root of $\mathcal{D}'$ and $Conn = V(H') \cap \bigcup \lambda(p)$ holds by the last condition of the definition of induced subhypergraphs. Hence, the foreach-loop on lines 24 – 39 is eventually entered. If function `Decomp` does not return "true" before, it will eventually try $\lambda(c)$ in the foreach-statement on line 24. Then $\chi(c)$ assigned on line 25 is the correct χ-label of c according to the normal form. The connectedness check on line 26 succeeds since $\mathcal{D}$ satisfies the connectedness condition. By assumption, c is a balanced separator; hence also the check on line 29 succeeds. Thus, the foreach-loop on lines 31 –34 is executed. It is easy to verify that the parameters supplied to `Decomp` in the recursive calls on line 33 correspond to induced subhypergraphs. Therefore, all these calls of `Decomp` return "true" by the induction hypothesis. Hence, also the statements on lines 35 – 37 are executed. In this case, since $comp_{down}$ comprises all edges and special edges of H', `Decomp` is called on line 37 with $comp_{up}.E = \emptyset$ and $comp_{up}.Sp = \{\chi(c)\}$. Hence, as was shown in the induction begin, this call of `Decomp` returns "true". Therefore, the return-statement on line 39 is executed and the overall result "true" is returned by function `Decomp`.

Case 2. Suppose that c is not the root node of $\mathcal{D}'$. Then c has a parent node inside $\mathcal{D}'$. Let us refer to this parent node as p. If function `Decomp` has not already returned "true" before, it will eventually try $\lambda(p)$ in the foreach-statement on line 16. By Corollary 4.8, one of the $[\lambda(p)]$-components of H' is the $[\chi(p)]$-component consisting of the edges and special edges which are covered in $\mathcal{D}$ by the subtree rooted at child node c of p. The check on line 18 is successful because the child c of p is a balanced separator. Hence c and the subtrees below c cover more than half of the edges and special edges. The check on line 22 succeeds because of the connectedness condition in $\mathcal{D}$. Hence, the foreach-loop on lines 24 – 39 is eventually entered. If function `Decomp` does not return "true" before, it will eventually try $\lambda(c)$ in the foreach-statement on line 24. Then $\chi(c)$ assigned on line 25 is the correct χ-label of c according to the normal form. The connectedness check on line 26 succeeds since $\mathcal{D}$ satisfies the connectedness condition. By assumption, c is a balanced separator; hence also the check on line 29 succeeds. Thus, the foreach-loop on lines 31 –34 is executed. It is easy to verify that the parameters supplied to `Decomp` in the recursive calls on line 33 correspond to induced subhypergraphs. Hence, all these calls of `Decomp` return "true" by the induction hypothesis. Hence, also the statements on lines 35 – 37 are executed. Again, it is easy to verify that also the parameters supplied to `Decomp` in the recursive call on line 37 correspond to an induced subhypergraph. Hence, by the induction hypothesis, also this call of `Decomp` returns "true". Therefore, the return-statement on line 39 is executed and the overall result "true" is returned by function `Decomp`. $\square$

4.4 An Illustrative Example

To illustrate the notions introduced in Section 4.1 and the basic algorithm `log-`k`-decomp` shown in Algorithm 4.1, we consider the hypergraph $H = (V, E)$ with $V = \{x_1, \ldots, x_{10}\}$ and

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

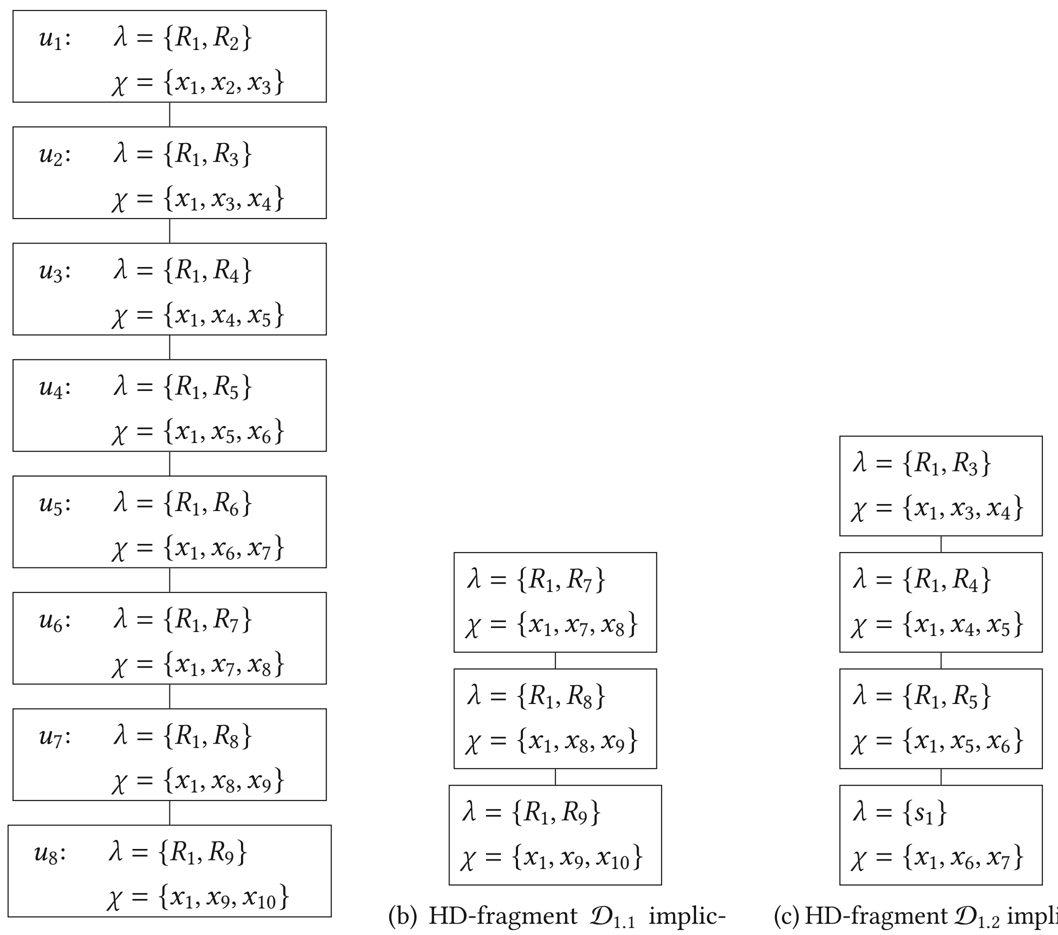
In other words, H is a essentially a cycle of size 10. A hypertree decomposition $\mathcal{D}$ of H is shown in Figure 4.1a.

We now walk through algorithm `log-`k`-decomp`, which will allow us to see also the notions from Section 4.1 in action. Suppose that we run `log-`k`-decomp` with hypergraph H and parameter $k = 2$. In each of the loops on lines 3, 16, and 24, the algorithm searches for a λ-label until it finds a successful one. By "successful" we mean that the current execution of the main program or of function `Decomp` returns true (on line 9 or 39, respectively). To keep things simple in our discussion below, we will directly choose a successful one with the understanding, that this particular λ-label will eventually be selected by the program unless another successful one has already been found before.

Main program. The RootLoop tests all possible candidates for λ_r. It will eventually try $\lambda_r = \{R_1, R_2\}$ (as mentioned above – with the understanding that it has not found another successful one before). There is only one $[\lambda_r]$-component y, namely $y.E = \{R_3, \ldots, R_{10}\}$ and $y.Sp = \emptyset$. Moreover, on line 6, we set $Conn_y = \{x_1, x_3\}$ and call function `Decomp` for this combination of component y and vertex set $Conn_y$.

Call 1 of function `Decomp` with parameters $H'.E = \{R_3, \ldots, R_{10}\}$, $H'.Sp = \emptyset$, and $Conn = \{x_1, x_3\}$. Since none of the conditions of the base case is satisfied, the ParentLoop will be entered. It will eventually try $\lambda_p = \{R_1, R_5\}$. This splits H' into 2 components $c_1 = \{R_3, R_4\}$ and $c_2 = \{R_6, R_7, R_8, R_9, R_{10}\}$. Component c_2 satisfies the size constraint on line 18 and, therefore, becomes $comp_{down}$ on line 19.

The ChildLoop will eventually try $\lambda_c = \{R_1, R_6\}$. On line 25, we thus set $\chi_c = \{x_1, x_6, x_7\}$. This gives rise to a single component $comps_p[i]$ inside $comp_{down}$, namely $comps_p[i] = \{R_7, R_8, R_9, R_{10}\}$. This component satisfies both the constraint for connectedness (line 26) and the size constraint


u_1: $\lambda = \{R_1, R_2\}$ $\chi = \{x_1, x_2, x_3\}$
u_2: $\lambda = \{R_1, R_3\}$ $\chi = \{x_1, x_3, x_4\}$
u_3: $\lambda = \{R_1, R_4\}$ $\chi = \{x_1, x_4, x_5\}$
u_4: $\lambda = \{R_1, R_5\}$ $\chi = \{x_1, x_5, x_6\}$
u_5: $\lambda = \{R_1, R_6\}$ $\chi = \{x_1, x_6, x_7\}$
u_6: $\lambda = \{R_1, R_7\}$ $\chi = \{x_1, x_7, x_8\}$
u_7: $\lambda = \{R_1, R_8\}$ $\chi = \{x_1, x_8, x_9\}$
u_8: $\lambda = \{R_1, R_9\}$ $\chi = \{x_1, x_9, x_{10}\}$
$\lambda = \{R_1, R_7\}$ $\chi = \{x_1, x_7, x_8\}$
$\lambda = \{R_1, R_8\}$ $\chi = \{x_1, x_8, x_9\}$
$\lambda = \{R_1, R_9\}$ $\chi = \{x_1, x_9, x_{10}\}$
$\lambda = \{R_1, R_3\}$ $\chi = \{x_1, x_3, x_4\}$
$\lambda = \{R_1, R_4\}$ $\chi = \{x_1, x_4, x_5\}$
$\lambda = \{R_1, R_5\}$ $\chi = \{x_1, x_5, x_6\}$
$\lambda = \{s_1\}$ $\chi = \{x_1, x_6, x_7\}$

(a) HD $\mathcal{D}$ of hypergraph H from Section 4.4

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

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

Figure 4.1: Visualisations of the HD constructed as part of Section 4.4 and the HD-fragments used for its construction.

(line 29). Moreover, it leads to 2 recursive calls of function `Decomp`: one for the component $comps_p[i] = \{R_7, R_8, R_9, R_{10}\}$ "below" the "child node" (on line 33) and one for the component $c_1 = \{R_3, R_4, R_5\}$ "above" (on line 37). For the component "below", we have $x.E = \{R_7, R_8, R_9, R_{10}\}$ and $x.Sp = \emptyset$; moreover, we set $Conn_x = \{x_1, x_7\}$ on line 32. For the component "above", we set $comp_{up}.E = \{R_3, R_4, R_5\}$ on line 35 and $comp_{up}.Sp = \{s_1\}$ with $s_1 = \chi_c = \{x_1, x_6, x_7\}$ on line 36; moreover, we take $Conn = \{x_1, x_3\}$ from the current call of `Decomp`. Clearly, edge R_6 from H' is already covered by $\chi_c = \{x_1, x_6, x_7\}$ and does not need to be further considered.

As we shall work out next, Call 1.1 of function `Decomp` for the component "below" will return true based on the HD-fragment $\mathcal{D}_{1.1}$ shown in Figure 4.1b. Likewise, Call 1.2 of function `Decomp` for the component "above" will return true based on the HD-fragment $\mathcal{D}_{1.2}$ shown in Figure 4.1c. The leaf node of $\mathcal{D}_{1.2}$ contains the special edge s_1, which acts as a placeholder for the node c

with labels $\lambda_c = \{R_1, R_6\}$ and $\chi_c = \{x_1, x_6, x_7\}$ from the current call of `Decomp`.

The HD-fragment $\mathcal{D}_1$ of the successful Call 1 of function `Decomp` is then obtained by taking HD-fragment $\mathcal{D}_{1.2}$, replacing the leaf node with λ-label $\{s_1\}$ by the node c with $\lambda_c = \{R_1, R_6\}$ and $\chi_c = \{x_1, x_6, x_7\}$ and appending the HD-fragment $\mathcal{D}_{1.1}$ below this node c. In other words, the HD-fragment $\mathcal{D}_1$ is precisely HD $\mathcal{D}$ minus its root node, i.e., nodes $u_2 - u_8$ of $\mathcal{D}$. The final HD $\mathcal{D}$ (shown in see Figure 4.1a) is obtained by combining the root node (whose λ-label was determined in the RootLoop of the main program, line 3) with HD-fragment $\mathcal{D}_1$.

Call 1.1 of function `Decomp` with parameters $H'.E = \{R_7, R_8, R_9, R_{10}\}$, $H'.Sp = \emptyset$, and $Conn = \{x_1, x_7\}$. The ParentLoop (line 16) and the ChildLoop (line 24) will eventually choose $\lambda_p = \{R_1, R_7\}$ and $\lambda_c = \{R_1, R_8\}$, respectively. Then function `Decomp` is called recursively with parameters $x.E = \{R_9, R_{10}\}$, $x.Sp = \emptyset$, and $Conn_x = \{x_1, x_9\}$ on line 33 for the only component "below" and with parameters $comp_{up}.E = \{R_7\}$, $comp_{up}.Sp = \{s_2\}$ with $s_2 = \{x_1, x_8, x_9\}$, and $Conn = \{x_1, x_7\}$ on line 37 for the component "above".

Call 1.1.1 of function `Decomp` with parameters $H'.E = \{R_9, R_{10}\}$, $H'.Sp = \emptyset$, and $Conn = \{x_1, x_9\}$. This call immediately returns true since we have reached the base case on lines 12 - 13. The corresponding HD-fragment $\mathcal{D}_{1.1.1}$ consists of a single node with λ-label $\{R_9, R_{10}\}$.

Call 1.1.2 of function `Decomp` with parameters $H'.E = \{R_7\}$, $H'.Sp = \{s_2\}$ with $s_2 = \{x_1, x_8, x_9\}$, and $Conn = \{x_1, x_7\}$. In the ParentLoop, eventually, $\lambda_p = \{R_1, R_6\}$ will be chosen on line 16. In this case, $comp_{down}$ is actually all of H' and it clearly satisfies the size constraint on line 18. In the ChildLoop, $\lambda_c = \{R_1, R_7\}$ will eventually be chosen on line 24. It gives rise to $\chi_c = \{x_1, x_7, x_8\}$ on line 25 with a single $[\chi_c]$-component $comps_c[i] = \{s_2\}$. Function `Decomp` will therefore be called on line 33 with parameters $x.E = \emptyset$, $x.Sp = \{s_2\}$, and $Conn_x = \{x_1, x_8\}$. This call (referred to as Call 1.1.2.1) returns true since we now have the base case on lines 14 - 15.

The recursive call of function `Decomp` on line 37 for the "components above" is a special case where there are actually no such "components above" left. Hence, in this case, we have $comp_{up}.E = \emptyset$, $comp_{up}.Sp = \{s_3\}$ with $s_3 = \chi_c = \{x_1, x_7, x_8\}$, and $Conn = \{x_1, x_7\}$. This leads to the Call 1.1.2.2 of function `Decomp`, which returns true since we again have the base case on lines 14 - 15.

In total, Call 1.1.2 of function `Decomp` is successful and the corresponding HD-fragment $\mathcal{D}_{1.1.2}$ consists of 2 nodes: the root node with λ-label $\{R_1, R_7\}$ and its child node with λ-label $\{s_2\}$

We can now also construct the HD-fragment $\mathcal{D}_{1.1}$ of the successful Call 1.1 of function `Decomp`. More precisely, HD-fragment $\mathcal{D}_{1.1}$ is obtained by taking HD-fragment $\mathcal{D}_{1.1.2}$, replacing the leaf node with λ-label $\{s_2\}$ by the node c with $\lambda_c = \{R_1, R_8\}$ and $\chi_c = \{x_1, x_8, x_9\}$ from Call 1.1 and appending the HD-fragment $\mathcal{D}_{1.1.1}$ below this node c. The resulting HD-fragment $\mathcal{D}_{1.1}$ is shown in Figure 4.1b. That is, $\mathcal{D}_{1.1}$ is the subtree consisting of the bottom 3 nodes u_6, u_7, and u_8, of the final HD $\mathcal{D}$ displayed in Figure 4.1a.

Call 1.2 of function `Decomp` with parameters $H'.E = \{R_3, R_4, R_5\}$, $H'.Sp = \{s_1\}$ with $s_1 = \{x_1, x_6, x_7\}$, and $Conn = \{x_1, x_3\}$. The execution of this function call is very similar to the calls discussed above. Below, we therefore do not discuss in detail the remaining recursive calls inside Call 1.2. Instead, we only list for each such call the parameters, the balanced separator χ_c, and the corresponding HD-fragments.

In Call 1.2 of function `Decomp`, eventually the balanced separator with $\lambda_c = \{R_1, R_4\}$ and $\chi_c = \{x_1, x_4, x_5\}$ will be chosen, which gives rise to the recursive Calls 1.2.1 (line 33) and 1.2.2 (line 37) of function `Decomp`, which we briefly discuss below.

Call 1.2.1 of function `Decomp` with parameters $H'.E = \{R_5\}$, $H'.Sp = \{s_1\}$ with $s_1 = \{x_1, x_6, x_7\}$, and $Conn = \{x_1, x_5\}$. As in the Call 1.1.2, we now have a connection subhypergraph of H consisting of a single edge and a single special edge. Analogously to Call 1.1.2, also Call 1.2.1 returns true and the corresponding HD-fragment $\mathcal{D}_{1.2.1}$ consists of 2 nodes: the root node with λ-label $\{R_1, R_5\}$ and its child node with λ-label $\{s_1\}$.

Call 1.2.2 of function `Decomp` with parameters $H'.E = \{R_3\}$, $H'.Sp = \{s_4\}$ with $s_4 = \{x_1, x_4, x_5\}$, and $Conn = \{x_1, x_3\}$. Again, we have a connection subhypergraph of H consisting of a single edge and a single special edge. Analogously to the Calls 1.1.2 and 1.2.1, also Call 1.2.2 returns true and the corresponding HD-fragment $\mathcal{D}_{1.2.2}$ consists of 2 nodes: the root node with λ-label $\{R_1, R_3\}$ and its child node with λ-label $\{s_4\}$.

We can now construct the HD-fragment $\mathcal{D}_{1.2}$ of the successful Call 1.2 by taking HD-fragment $\mathcal{D}_{1.2.2}$, replacing the leaf node with λ-label $\{s_4\}$ by the node c with $\lambda_c = \{R_1, R_4\}$ and $\chi_c = \{x_1, x_4, x_5\}$ from Call 1.2 and appending the HD-fragment $\mathcal{D}_{1.2.1}$ below this node c. The resulting HD-fragment $\mathcal{D}_{1.2}$ is shown in Figure 4.1c.

4.5 Further Combinatorial Observations and Optimisations

As was shown in Theorem 4.11, algorithm `log-`k`-decomp` introduced in Section 4.2 reaches the primary goal of splitting the HD-construction into subtasks with guaranteed upper bound on their size. In theory, this is enough to support parallelism. However, this basic algorithm still leaves a lot of room for further improvements. In this section, we present several optimisations, which are crucial to achieve good performance in practice. The line numbers below refer to Algorithm 4.1. However, in Algorithm 4.2, we will ultimately also give the pseudo-code for the enhanced algorithm where all the optimisations mentioned below are included.

Extension of the base case. The recursive function `Decomp` starts (on lines 12 – 15) with some simple checks that immediately give a “true” answer. In contrast, a “false” answer is only obtained in case of unsuccessful execution of the entire procedure. We could add the following negative case to the top of the procedure: if $H'.E = \emptyset$, then $|H'.Sp| \leq 1$ must hold. The rationale of this condition is that, if there are no more edges in $H'.E$, then we would have to use only “old” edges (i.e., edges covered already at some node further up in the HD) in the λ-label to separate the remaining special edges. However, a λ-label consisting of “old” edges only is not allowed, since this would violate the second condition of the normal form in Definition 4.5 (i.e., "some progress has to be made").

Root of the HD-fragment. In the current form of procedure `Decomp`, we always “guess” a pair (p, c) of nodes, such that p is the parent of c. This also covers the case that c is the root node of the HD-fragment for the current connection subhypergraph. In this case, the parent node p would actually be the node immediately above this HD-fragment (in other words, p was the node from which the current call of `Decomp` happened). However, it would be more efficient to consider the case of “guessing” the root node of this HD-fragment explicitly. More precisely, we would thus first check for the label λ_p guessed in `Decomp` on line 16 (which, in the current version of the algorithm, is automatically treated as the “parent”) if all $[\lambda_p]$-components have at most half the size of the current connection subhypergraph.
If this is the case, then we may use this node as the root of the HD-fragment to cover the current connection subhypergraph. This makes sense since it corresponds precisely to the “search” for a balanced separator in the proof of Lemma 4.10. That is, if the root of the HD gives rise to components which are all at most half the size, then the root *is* the desired balanced separator. If this is not the case, then we simply proceed with procedure `Decomp` in its present form, i.e.: there exists exactly one $[\lambda_p]$-component whose size is bigger than half. So we take the guessed node as the parent and search for a balanced separator as a child of p in the direction of this oversized $[\lambda_p]$-component.

Allowed edges. The main task of procedure `Decomp` is to compute labels (i.e., edge sets) λ_p and λ_c of nodes p, c, which will ultimately be in a parent-child relationship in the HD. For these labels, Algorithm 4.1 imposes no restriction. That is, in principle, we would try all possible sets of $\leq k$ edges for these labels. However, not all edges actually make sense. We should thus add one more parameter to procedure `Decomp` indicating the edges that are allowed in a λ-label of the HD-fragment for this connection subhypergraph.
More specifically, in our search for the λ-label of some node u, we may exclude from the HD of the connection subhypergraph $comp_{up}$ (i.e., in the recursive call of function `Decomp` on line 37) all edges which are part of some component “below” u. The rationale of this restriction is that, by the special condition, using a “new” edge in a λ-label forces us to add all its vertices to the χ-label, i.e.: it is fully covered in such a node. But then it cannot be part of a component whose edges are covered for the first time further down in the tree.
Note that we can yet further restrict the search for the label λ_c by requiring that at least one edge must be from $H'.E$, since choosing only “old” edges would violate the second condition of the normal form. As far as the label λ_p is concerned, the same kind of restriction can be applied if we first implement the previous optimisation of handling the root node of the current HD-fragment separately. If we indeed have to guess the labels λ_p and λ_c of *two* nodes p and c (i.e., the label λ_p guessed first was not a balanced separator), then both nodes p and c are *inside* the current HD-fragment. Hence, also the label λ_p must contain at least one “new” edge.

No special treatment of the root of the HD. In the current form of the algorithm, we start in the main program by “guessing” the label λ_r of the root of the HD on line 3 and then branch into calls of procedure `Decomp` for each $[\lambda_r]$-component. Of course, there is no guarantee that this λ-label is a balanced separator. Consequently, there is no guarantee that the size of *all* HD-fragments to be constructed in these calls of procedure `Decomp` is significantly smaller than

the entire HD.
In order to start with a balanced separator right from the beginning, we may instead call procedure `Decomp` straight away with parameters $H'.E = H$, $H'.Sp = \emptyset$, and $Conn = \emptyset$. We thus treat the search for the very first λ-label in the desired HD in exactly the same way as for any other HD-fragment. As far as the above mentioned optimisation of restricting the allowed edges is concerned, of course all edges of H would be initially allowed.

Searching for child nodes first. In Algorithm 4.1, we first look for λ-labels of potential parent nodes, and consider afterwards the λ-labels of potential child nodes. Only then do we check if the χ-label of the child is a balanced separator of the current subcomponent. We have observed that in many hypergraphs of HyperBench, balanced separators are rare, in the sense that only a small part of the search space will ever fulfil the properties required. Therefore we should first look for a potential child s.t. its λ-label is a balanced separator, and only afterwards try to find a fitting parent. While this may seem slightly unintuitive, it allows us to quickly detect cases where no balanced separator can be found at all.
Note that we can determine the precise bag χ_c for a child c only when we know the λ-label of its parent. Nevertheless, even if we only have λ_c, we can over-approximate the χ_c-label as $\bigcup \lambda_c$. Hence, if $\bigcup \lambda_c$ is not a balanced separator, then we may clearly conclude that neither is χ_c.
Finally, note that by searching for the child node first, we get the above described optimisation of treating the "Root of the HD-fragment" separately almost for free. Indeed, when computing λ_c, we can immediately check if $Conn \subseteq \bigcup \lambda_c$ holds. Recall that $Conn$ constitutes the interface to the HD-fragment *above* the current one. Hence, if $\bigcup \lambda_c$ fully covers this interface, c is in fact the root node of the current HD-fragment.

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

Theorem 4.12. *The optimised `log-k-decomp` algorithm for checking if a hypergraph H has $hw(H) \leq k$ given in Algorithm 4.2 is sound and complete. More specifically, for given hypergraph H and integer $k \geq 1$, the algorithm returns "true" if and only if there exists an HD of H of width $\leq k$. Moreover, by materialising the decompositions implicitly constructed in the recursive calls of the `Decomp` function, an HD of H of width $\leq k$ can be constructed in polynomial time in case of a successful computation (i.e., return-value "true").*

Proof. The soundness and completeness of Algorithm 4.2 follow almost immediately from the soundness and completeness of Algorithm 4.1 together with the above explanations of the various optimisations. Likewise, the polynomial-time upper bound on the time needed to construct an HD in case of a successful computation can again be easily shown as part of the soundness proof. The only non-trivial part is that the last optimisation (i.e., the restriction of the search space for $\lambda(p)$) does not destroy the completeness of the algorithm. The remainder of the proof will concentrate on this aspect.

Assume that hypergraph H has an HD of width $\leq k$. Then the optimised `log-`k`-decomp` algorithm without the restriction on the search space for label λ_p (on line 22) returns the overall result *true*. This is due to the fact that, as was argued in Section 4.5, the other optimisations mentioned there do not affect the completeness of the algorithm. Now consider a recursive call of function `Decomp` and suppose that it returns true if the restriction on the search space for label λ_p is dropped. Of course, if the value *true* is returned in one of the base cases (lines 6 or 8) or if λ_c turns out to be the λ-label of the root node of the current HD-fragment (and *true* is returned on line 21), then the restriction of the search space for λ_p has no effect at all. Hence, the only interesting case to consider is that the parent loop (lines 22 – 43) is indeed executed.

Let λ_p be the λ-label chosen on line 22 if no restriction is imposed on the search space. We claim that we may remove from λ_p all edges that have an empty intersection width $\bigcup \lambda_c$ without altering the control flow of this particular execution of function `Decomp`. Actually, it suffices to show that we may remove *one* edge e with an empty intersection width $\bigcup \lambda_c$ from λ_p without altering the control flow of this particular execution of function `Decomp`. Then the claim follows by an easy induction argument. f

So suppose that λ_p contains at least one edge e such that $e \cap \bigcup \lambda_c = \emptyset$ and let $\lambda'_p = \lambda_p \setminus \{e\}$. An inspection of the code of the parent loop reveals that it suffices to show that this elimination of edge e from λ_p leaves $comp_{down}$ unchanged. Indeed, if $comp_{down}$ is still a $[\lambda'_p]$-component, say the i-th $[\lambda'_p]$-component, then the if-condition on line 24 is true. Of course, there can be only one $[\lambda'_p]$-component satisfying the condition $comps_p[i]| > \frac{|H'|}{2}$. Hence, on line 25, for this particular i, exactly the same value is assigned to $comp_{down}$ for λ'_p as for λ_p. But then also χ_c on line 28 gets the same value as without the restriction on the search space of λ_p. Consequently, also the $[\chi_c]$-components computed on line 33 and the parameters supplied to the recursive calls of function `Decomp` (on lines 36 and 41) remain the same as without the restriction on the search space. Hence, function `Decomp` will ultimately return the value *true* also if we choose λ'_p on line 22.

It remains to show that λ_p and λ'_p indeed give rise to the same component $comp_{down}$. To avoid confusion, let us write $comp_{down}$ to denote a $[\lambda_p]$-component and $comp'_{down}$ to denote a $[\lambda'_p]$-component. Let $comp_{down}$ be the unique $[\lambda_p]$-component that satisfies the condition $|comps_p[i]| > \frac{|H'|}{2}$ on line 24. We have $\lambda'_p \subseteq \lambda_p$. Decreasing a set can only increase the corresponding components. Hence, there exists a $[\lambda'_p]$-component, call it $comp'_{down}$ with $comp_{down} \subseteq comp'_{down}$. We have to show that $comp_{down} = comp'_{down}$ holds.

The set $comp_{down}$ consists of the edges and special edges of the $[\chi_c]$-components contained in $comp_{down}$ (denoted as $comps_c$ in the algorithm), and the edges and special edges covered by χ_c.

Let us refer to these $[\chi_c]$-components as $C_1, \ldots, C_\ell$. By Corollary 4.8, these $[\chi(c)]$-components are at the same time the $[\lambda_c]$components contained in $comp_{down}$. And the edges and special edges covered by χ_c are of course also covered by $\bigcup \lambda_c$. Likewise, $comp'_{down}$ consists of the (special) edges of the $[\lambda_c]$components contained in $comp'_{down}$ plus the (special) edges covered by λ_c.

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

Since all (special) edges in $comp'_{down}$ are either in some $[\lambda_c]$-component contained in $comp'_{down}$ or covered by $\bigcup \lambda_c$, and since we are assuming that π is of minimal length, the path π starts with f in some $[\lambda_c]$-component C_i, possibly goes through $\bigcup \lambda_c$ and ends with f' in component C'. Recall that e was chosen such that $e \cap \bigcup \lambda_c = \emptyset$. Hence, the edges f_α and $f_{\alpha+1}$ cannot be covered by $\bigcup \lambda_c$. By our assumption that π has minimal length, we can also exclude the case that both f_α and $f_{\alpha+1}$ are in C'. Hence, at least one of f_α and $f_{\alpha+1}$ must be in C_i. In other words, $e \cap C_i \neq \emptyset$. Hence, also $e \cap comp_{down} \neq \emptyset$. However, by the check on line 31 in Algorithm 4.2, we know that $V(comp_{down}) \cap \bigcup \lambda_p \subseteq \bigcup \lambda_c$. This contradicts the assumption that $e \in \lambda_p$ and $e \cap \bigcup \lambda_c = \emptyset$. □

4.6 Implementation and Evaluation

We report now on the empirical results obtained for our implementation of the `log-`k`-decomp` algorithm. Our experiments are based on the HyperBench benchmark from [25], which was already used for the evaluation of previous decomposition algorithms, notably `NewDetKDecomp` [25] (an enhanced re-implementation of `det-`k`-decomp` [50]) and `HtdLEO` [86].

Our goal was to determine the exact hypertree width of as many instances as possible. We compare here the performance of three different decomposition methods, namely `NewDetKDecomp` [25], `HtdLEO` [86], and our implementation of `log-`k`-decomp`. Note that while the tested implementations include the capability to compute GHDs or FHDs, we only consider the computation of HDs in our experiments here. Our new implementation of `log-`k`-decomp` is based on the open-source code of BalancedGo, the parallel algorithm for computing GHDs we presented in Chapter 3 of this thesis.

The full raw data of our experiments is publicly available [36], as is the source code of our implementation[1] of `log-`k`-decomp`.

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

Algorithm 4.2: Optimised `log-k-decomp`

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

4.6.1 Benchmark Instances and Setting

For the evaluation, we use the benchmark library HyperBench [25]. It contains 3648 hypergraphs underlying CQs and CSPS from various sources in industry and the literature and is commonly used to evaluate decomposition algorithms. The instances are available at `http://hyperbench.dbai.tuwien.ac.at` and also in the published raw data of our experiments [36].

Hardware and Software. Our implementation is written in the programming language Go using version 1.14 and we will refer to it as `log-`k`-decomp`. We will give more details below on how it was configured for the experiments reported in Section 4.6.2. The hardware used for the evaluation was a cluster of 12 nodes, using Ubuntu 16.04.1 LTS, with Linux kernel 4.4.0-184-generic, GCC version 5.4.0. Each node has a 12 core Intel Xeon CPU E5-2650 v4, clocked at 2.20 GHz and using 264 GB of RAM.

Setup of Experiments. To ensure comparability of our experiments with results published in the literature, we employ the following test setup and restrictions: a timeout of one hour was used and available RAM was limited to 1 GB. We note that this corresponds to limits also used in previous experiments in this area [25] and is indeed the same as we used in the experimental evaluation section of Chapter 3. For `log-`k`-decomp`, each run needs two inputs: a hypergraph H and the width parameter $k \geq 1$. For these tests, we used width parameters in the range $[1, 10]$. When running tests for `HtdLEO`, we used different memory limits. Namely, we allowed `HtdLEO` to use up to 24 GB of RAM since SMT solving is significantly more memory intensive than the other two algorithms. Note that the other two algorithms have very low memory requirements and are not constrained in any way by the 1 GB limit and the respective experiments are therefore still comparable with `HtdLEO`. Furthermore, `HtdLEO` needs no width parameter since it directly tries to find an optimal solution.

We used the HTCondor system [88] to facilitate the tests, limits to memory and number of cores accessed by running test instances.

Throughout this section we will be interested in two key metrics. First, the number of *solved* instances, by which we mean instances for which an optimal (i.e., minimal width) hypertree decomposition was found and proven optimal. Second, the computation time that was necessary to compute the optimal width decomposition, which we will refer to as the *running time* or simply *runtime*. Importantly, this means that average running times are taken only over the instances that the respective algorithm is able to solve, while timed out instances are not considered in the running time calculation.

4.6.2 Empirical Evaluation

We report here on the main results of our experiments. A number of additional experiments can be found in Appendix 4.7, providing a variety of further details and insights. Our implementation of `log-`k`-decomp` also employs the following *hybridisation strategy*: as will be seen below, `NewDetKDecomp` performs very well on small hypergraphs but has difficulties with even slightly larger instances. In contrast, a particular strength of our new `log-`k`-decomp` algorithm is to

Table 4.1: Comparison of prior methods and `log-`k`-decomp`: number of cases solved and runtimes (sec.) to find optimal-width HDs.

			Hypertree Decomposition Methods							
			`NewDetKDecomp` [25]				`HtdLEO` [86]			
Origin of Instances	Size of Instances	Instances in Group	#solved	avg	max	stdev	#solved	avg	max	stdev
Application	$75 < \|E\| \leq 100$	405	97	21.4	3296.0	192.8	65	809.5	3156.6	735.2
	$50 < \|E\| \leq 75$	514	276	10.6	1906.0	104.7	448	250.0	3281.5	409.3
	$10 < \|E\| \leq 50$	369	253	0.0	0.0	0.0	237	60.1	1017.9	150.3
	$\|E\| \leq 10$	915	906	0.0	0.0	0.0	876	56.6	1427.1	155.0
Synthetic	$\|E\| > 100$	66	18	0.2	7.0	1.0	13	734.0	2507.1	711.7
	$75 < \|E\| \leq 100$	422	87	77.2	3467.0	379.3	**312**	1045.2	3591.1	1287.0
	$50 < \|E\| \leq 75$	215	38	18.8	1593.0	141.9	212	101.7	2560.1	246.1
	$10 < \|E\| \leq 50$	647	290	56.0	3240.0	336.3	303	412.2	3597.4	850.2
	$\|E\| \leq 10$	95	**95**	0.0	0.0	0.0	78	28.8	218.5	41.5
Total	-	3648	2060	20.6	3467.0	194.2	2544	280.2	3597.4	676.7

			Hypertree Decomposition Methods			
			`log-`k`-decomp` Hybrid			
Origin of Instances	Size of Instances	Instances in Group	#solved	avg	max	stdev
Application	$75 < \|E\| \leq 100$	405	**261**	86.5	3555.8	332.4
	$50 < \|E\| \leq 75$	514	**469**	0.5	78.5	3.6
	$10 < \|E\| \leq 50$	369	**253**	0.0	0.1	0.0
	$\|E\| \leq 10$	915	**915**	0.0	0.0	0.0
Synthetic	$\|E\| > 100$	66	**34**	46.9	2528.2	209.6
	$75 < \|E\| \leq 100$	422	235	48.9	2495.6	210.9
	$50 < \|E\| \leq 75$	215	**215**	4.1	476.3	32.7
	$10 < \|E\| \leq 50$	647	**625**	18.8	3526.3	174.7
	$\|E\| \leq 10$	95	**95**	0.0	0.0	0.0
Total	-	3648	**3102**	30.5	3555.8	197.8

quickly split a big hypergraph into significantly smaller connection subhypergraphs. To combine the best of both worlds, we use `log-`k`-decomp` to split the original HD computation problem until the subproblems become small, at which point we apply our own implementation of `det-`k`-decomp` (extended to handle connection subhypergraphs correctly) to the small subproblems. For details and an experimental evaluation of different parametrisations for our hybridisation strategy, see Appendix 4.7.2.

We compare the aforementioned hybrid version of the `log-`k`-decomp` algorithm with the two state-of-the-art implementations for finding HDs: `NewDetKDecomp` [25] and `HtdLEO` [86].

Our results are summarised in Table 4.1, distinguishing the hypergraphs in the HyperBench benchmark by size and origin [2]. We distinguish between two main categories, hypergraphs that are derived from applications and hypergraphs that were synthetically generated. In each group we report our results split by the number of edges $|E|$ in the instance. Note that the group $|E| > 100$ of instances with more than 100 edges is empty for the Application case and thus omitted from the table. *Instances in Group* reports the number of instances in each such group. For each algorithm and each group of instances, we list the number of solved instances (*#solved*) and statistics over the running times (*avg*, *max*, *stdev*). Times are all in seconds and rounded to a single digit after the comma. Results over all groups are given in the last row titled "Total".

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

It may be of further interest how these numbers compare to the performance of state of the art algorithms for finding generalised hypertree decompositions. The results reported for BalancedGo, presented in Chapter 3 of this thesis, (on the same system) show that the best method there solves only 1730 instances optimally without timeout. In contrast `log-`k`-decomp` manages to solve 2491 of the instances tested there optimally[3]. Furthermore, in none of the cases where BalancedGo finds the optimal *ghw* is it lower than the optimal *hw*. In other words, in practice, the additional complexity of GHDs compared with HDs is not compensated by achieving lower width (even if, in theory, no better upper bound on the *hw* than $hw \leq 3 \cdot ghw + 1$ is known [4]).

In our experiments, we also observe that for low widths – i.e., cases where using HDs is most promising in practice – `log-`k`-decomp` is very close to solving all instances. In particular, of the 3224 instances with width at most 6, `log-`k`-decomp` solves 2930 (92%) instances. In contrast, `NewDetKDecomp` and `HtdLEO` time out on 1206 and 766, respectively, of those instances. This suggests that `log-`k`-decomp` can be a solid foundation for the integration of HDs in practice going forward. If we look at instances of $hw \leq 5$ the situation improves even further, with `log-`k`-decomp` solving 2450 out of 2482 (98.7%) instances; compared to 80% and 86% solved by `NewDetKDecomp` and `HtdLEO`, respectively.

The experiments reported in Table 4.1 were performed over the full set of HyperBench instances. However, for the additional experiments reported in this section, it is more meaningful to restrict our experiments to exclude hypergraphs that are, roughly speaking, too small or have high width. Small instances benefit only marginally from algorithmic improvements or parallelism,

[2]HyperBench instances are often categorised more fine-grained in terms of their origin (cf., [25]). For our experiments we have found the direct effect of hypergraph size to be more informative and therefore report our results in this way instead.

[3]The evaluation in [47] considers only a subset of HyperBench with 3071 instances

while very high width is of less algorithmic interest as it exponentially effects algorithms that make use of decompositions. Hence, we propose to exclude such instances to make more relevant observations. We therefore focus on instances with more than 50 edges and vertices that are known to have hypertree width at most 6. There are 465 instances in HyperBench which satisfy these conditions; we will refer to them as HB_{large}.

We performed a second set of experiments over the instances in HB_{large} to verify our claims that `log-`k`-decomp` is well-suited for parallelisation. For $1 \leq n \leq 5$, we observe the time taken to find and verify the optimal width of an instance using n CPU cores. We report on the times to find these optimum widths averaged over all instances in HB_{large} in Figure 4.2. To avoid a decreasing number of timeouts from skewing the data we report the average only over instances that do not timeout for any n for a given algorithm. For reference, we also report the (single core) performance of `NewDetKDecomp` for the same setting.

We observe approximately linear speedups up to 4 cores, from about 189 seconds on 1 core to 50 seconds for 4 cores for `log-`k`-decomp`. This behaviour is expected since our parallelisation strategy relies on dividing up the search space for bounded separators uniformly over the the available cores. Since this requires no communication between threads or other overhead that depends on the degree of parallelisation, the key task of searching for balanced separators scales linearly in the number of cores. In instances where the search for separators dominates the running time, such as negative instances where the full search space is explored, analysis of our algorithm therefore predicts effectively linear scaling of performance. In the data from Figure 4.2, we observe diminishing returns in the rate of improvement of average running time starting from 5 cores. However, preliminary experiments on additional different systems do not confirm this behaviour and there `log-`k`-decomp` exhibits linear scaling up to much higher core counts. Further in-depth experimentation is therefore required to obtain a clearer picture for the scaling behaviour for a high number of cores.

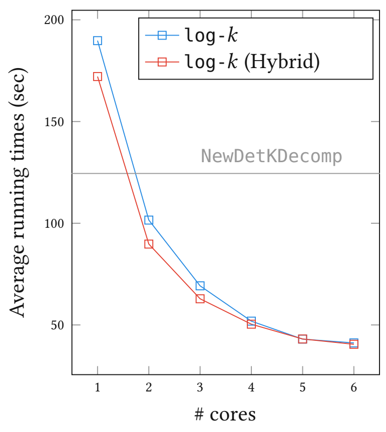
Very similar scaling can be observed for our Hybrid version. Note that the reported times for the Hybrid algorithm are slightly higher only due to solving more (harder) instances.

4.7 Additional Experimental Evaluation

We provide here a number of additional details on our implementation, such as the hybridisation strategy employed in our implementation of `log-`k`-decomp`, as well as further experiments to highlight various properties of our contribution.

4.7.1 Parallel Implementation

For our experiments, we implemented `log-`k`-decomp` including all of the optimisations presented in Section 4.5. As discussed above, a crucial aspect of our algorithm design is that the use of balanced separators allows us to recursively split the problem into smaller subproblems. The subproblems are independent of each other and are therefore processed in parallel by our implementation. Furthermore, following observations we made in Chapter 3, our implementa-



Method	Timeouts
`log`-k (Hybrid)	**143**
`log`-k	666
`NewDetKDecomp`	611

Figure 4.2: Study of `log`-k-`decomp` scaling behaviour w.r.t. the number of processing cores used.

tion also executes the search for balanced separators in parallel by partitioning the search space effectively.

4.7.2 Hybrid Approaches

While our algorithm has desirable properties for parallelisation (as has been pointed out in Section 4.7.1 above), this comes at the cost of some overhead when compared to simpler methods, in particular `det`-k-`decomp`. Especially on small and simple instances the restriction to balanced separators may act as a detriment to performance that outweighs its benefits for parallelisation and its effect on severely restricting the search space.

To balance these considerations overall in practice we therefore also consider hybrid variants of our implementation. Intuitively, we want to use `log`-k-`decomp` as long as the subproblems are still complex, but once they become simple, we want to switch to an algorithm that is better suited for those cases. For the simpler algorithm, `det`-k-`decomp` is the natural choice as it performs very well on small instances as was shown in [25], where an implementation of `det`-k-`decomp` is provided as part of `NewDetKDecomp`. To determine when the switch is made, we implemented two simple metrics to capture the complexity of a hypergraph:

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

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

Table 4.2: Study of two Hybrid methods of `log-`k`-decomp` on HB$_{large}$, and a comparison with `NewDetKDecomp` and `HtdLEO` for reference.

Method	Threshold	Solved	Av. runtime (sec.)
`WeightedCount`	200	395	92.15
`WeightedCount`	400	**411**	93.53
`WeightedCount`	600	410	**87.86**
`EdgeCount`	20	171	130.0
`EdgeCount`	40	219	145.09
`EdgeCount`	80	292	117.33
`NewDetKDecomp` [25]	-	174	318.93
`HtdLEO` [86]	-	277	779.39

We investigated the effectiveness of these metrics through a series of experiments. In particular, we ran experiments with both metrics and different thresholds for when to switch from `log-`k`-decomp` to `det-`k`-decomp`. To be precise, for a metric m and threshold T, `log-`k`-decomp` is executed for a subproblem with hypergraph H_i as long as $m(H_i) \geq T$. If $m(H_i) < T$, we switch to an implementation of `det-`k`-decomp` written from scratch as part of the code base of `log-`k`-decomp`. A similar strategy was already proposed in this thesis in Chapter 3 when designing the hybrid algorithm presented in that chapter.

However, in that system no metric for the complexity of a subproblem was employed, but rather the switch to `det-`k`-decomp` was always performed at a fixed recursion depth.

A significant portion of the hypergraphs in HyperBench are relatively small, so that even the full problem would be too simple for our metrics to be above reasonable thresholds. In those cases, the execution would be equivalent to simply running `det-`k`-decomp` on the instance. We therefore exclude small hypergraphs from these experiments to obtain more meaningful results by considering only the HB$_{large}$ instances here.

Our results for the experiments on HB$_{large}$ are summarised in Table 4.2. Methods `Weighted-Count` and `EdgeCount` refer to `log-`k`-decomp` with the respective metric used for hybridisation. The threshold column refers to parameter T in the discussion above. The experiments for all `log-`k`-decomp` hybrid methods had access to 12 cores, the experiments for `NewDetKDecomp` and `HtdLEO` used only 1 core each since they do not support parallelism.

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

One surprising observation from the table is that the differences in performance between different thresholds are much smaller for `WeightedCount` than for `EdgeCount`. Further investigation suggests that this is due to the WeightedCount metric decreasing much more rapidly as hypergraphs become simpler. At the same time, for subproblems that fall in the range between 200-600, the performance of switching to det-k-decomp immediately is roughly the same (on average) as the performance of continuing with log-k-decomp for one or two more steps.

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

4.7.3 HtdLEO with 10 Hour Timeout

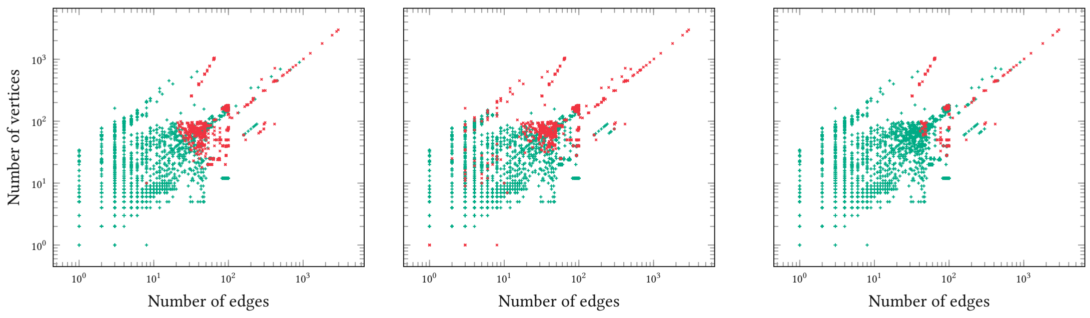
As mentioned before, the method used in HtdLEO fundamentally differs from the search algorithm here. In HtdLEO, the problem is encoded to the SAT modulo theories (SMT) setting and the final solving step is handed off to standard SMT solvers. The encoding in HtdLEO is constructed in such a way that no width parameter is handed to the solver, rather the encoding will always return the optimal width as its solution. This is a significant difference to the parameterised search implemented in log-k-decomp (but also previous algorithms such as det-k-decomp and BalancedGo).

This difference makes it naturally difficult to compare running times and the number of optimal solutions directly. To provide a fuller picture we add here Table 4.5 to provide a fuller picture. In the table we present the results from running the same experiments with HtdLEO but with the timeout increased to 10 hours. This increase in timeouts naturally makes the average runtimes difficult to compare to the values in Table 4.1. However, importantly we see that the increase in solved instances is also moderate, and overall log-k-decomp still solves significantly more instances than HtdLEO with 10 hours of maximum running time.

4.7.4 Analysis on Scaling by Size

To gain further insight into the performance of each algorithm with respect to solving instances optimally, we investigate the size of solved and unsolved instances. To this end, Figure 4.3 provides (logarithmic) scatter plots for each of the three algorithms in our tests. In each plot, each instance is positioned according to its number of vertices and edges. Solved instances for each algorithm are drawn in green while unsolved instances are drawn in red.

The plots show that our intuition holds true in that solving large instances (in both axis) significantly benefits from using log-k-decomp. Most of the remaining hypergraphs are either extremely large, containing thousands of edges and vertices, or belong to very specific CSP classes which we know to have very high width (significantly beyond the width 10 limit used in our experiments) through graph-theoretic arguments.



(a) `NewDetKDecomp` (b) `HtdLEO` (c) `log-`*k*`-decomp`

Figure 4.3: Comparison of solved instances (green) and unsolved instances (red), relative to their edge and vertex size.

Table 4.3: Comparison of the decomposition methods by how many instances were solved for a specific width.

Width	Virtual Best	`NewDetKDecomp`	`HtdLEO`	`log-`*k*`-decomp`
1	709	677	649	**709**
2	595	586	567	**595**
3	310	**310**	273	**310**
4	386	379	321	**386**
5	450	38	341	**450**
6	485	28	307	**480**
7	124	9	16	**108**
8	115	1	**69**	46
9	19	0	1	**18**

4.7.5 Analysis on Determining Low Width

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

For this purpose, we first need the concept of the "Virtual Best" method. This notion simply aggregates the results of all other methods and shows how for how many instances of Hyper-Bench we know their *hw*. We can see in Table 4.3 how each of the three methods fares when compared against this virtual best method. For widths up to 5, the Hybrid `log-`*k*`-decomp` is unbeaten, solving all known instances, as well as solving many of them exclusively.

To provide a more detailed analysis, we also compare for how many instances of *hw* up to 6,

Table 4.4: Comparison of the decomposition methods by the upper bounds it could provide. Note that `HtdLEO` is not being explicitly considered here, since it directly computes the optimal width. Thus it would have the number 2544 — its number of solved instances – in each row.

Problem to solve	Virtual Best	`log`-k-`decomp` (Hybrid)	`NewDetKDecomp`	`log`-k-`decomp`
$hw \leq 1$	3648	**3648**	3616[4]	**3648**
$hw \leq 2$	3648	**3648**	3631	**3648**
$hw \leq 3$	3637	**3637**	3355	3567
$hw \leq 4$	3623	**3623**	2391	3178
$hw \leq 5$	3616	**3611**	2485	2924
$hw \leq 6$	3370	**3253**	2897	2349

Table 4.5: Extension of Table 4.1, with running times for `HtdLEO` extended to 10 hours

Origin of Instances	Size of Instances	Instances in Group	`HtdLEO` [86] 10 Hour Run #solved	Changes 1 hour run
Application	$75 < \|E\| \leq 100$	405	94	+ 29
	$50 < \|E\| \leq 75$	514	461	+ 13
	$10 < \|E\| \leq 50$	369	237	± 0
	$\|E\| \leq 10$	915	876	± 0
Synthetic	$\|E\| > 100$	66	13	± 0
	$75 < \|E\| \leq 100$	422	360	+ 48
	$50 < \|E\| \leq 75$	215	214	+ 2
	$10 < \|E\| \leq 50$	647	433	+130
	$\|E\| \leq 10$	95	78	± 0
Total	-	3648	2766	+222

each method can determine whether an instance has hw of lower than the given number or not, by finding an HD of such a width or determining that no such HD can exist. Note that this does not require proving optimality. We can see the results in Table 4.4. We can see that both `log`-k-`decomp` and `log`-k-`decomp` (Hybrid) are very good at this, with the Hybrid determining for 3253 (or almost 90% of) instances whether they have $hw \leq 6$. If we limit ourselves to $hw \leq 5$, it determines the question for 3611 or almost 99% of instances.

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

4.8 Summary

In this chapter we introduced a novel algorithm `log-`k`-decomp` for computing hypertree decompositions. Based on new theoretical insights and results on HDs, we were able to propose an algorithm that constructs decompositions in arbitrary order (rather than, e.g., in a strict top-down manner) while achieving a *balanced* separation into subproblems. In this way, we have obtained a logarithmic bound on the recursion depth of our algorithm, making it particularly well suited for parallelisation. We evaluated an implementation of `log-`k`-decomp` through experimental comparison with the state of the art. On the standard benchmark for hypertree decomposition HyperBench [25], we are able to achieve clear improvements both in the number of solved instances and in the time required to solve them.

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

Chapter 5

A Distributed Algorithm for Hypergraph Decompositions

For this third line of research, we focus on another class of algorithm for computing hypergraph decomposition. Instead of a top-down algorithm, as we had seen in Chapter 3 and in Chapter 4, we will present a method that produces hypergraph decompositions in a bottom-up fashion, as we shall explain. We motivate this new approach via two observations. The first observation is the presence of very challenging instances in the HyperBench dataset. To the best of our knowledge, these large hypergraphs cannot be optimally solved by any of existing decomposition methods in the literature. The second observation is the fact that the most powerful computer systems today are not actually single, independent machines with a shared memory architecture. Instead, the most powerful computer systems are clusters of computers, which introduce another property of computation: instead of merely requiring parallelisation, which allows to effectively use multiple cores of the same machine to solve the same problem, we need algorithms that can be *distributed* over multiple shared memory machines, necessitating communication between the parts of the algorithm that run on different machines.

The concept of *candidate tree decompositions* [39] (CTD) from Gottlob et al. [39] proved to be a good fit to our aim of designing a distributed algorithm. As we will explain in this chapter, it enables us to split the work of computing bags, the building *blocks* of all possible decompositions, from the task of finding a decomposition once enough blocks have been found. We shall note here that the application of the candidate tree framework to the distributed setting is novel work, and not at all straightforward. The authors of the cited paper introduce CTDs as a general tool to argue about the complexity of computing different kinds of hypergraph decomposition.

After introducing the necessary concepts, we proceed to present the distributed algorithm we designed. It consists of three individual programs which are meant to be run on separate machines inside the same cluster. The first program acts as a coordinator that guides the overall search, it starts the search and will receive the final decomposition once one is found. It does not do any computational work. The second program implements the search for new bags which can be used to form a decomposition. The search for new bags has the property that it can actually be run on many machines at once, allowing for multiple instantiations of it. The coordinator program will dynamically adjust and split the search space accordingly. The last program of our distributed algorithm implements the actual search for a decomposition based on the bags founds so far.

We then conclude this chapter by presenting experimental data of our prototype implementation. We used the Google Cloud Platform (GCP) as the distributed platform to run our algorithm. The built-in message passing system of the GCP, called PubSub [65], proved to be useful to facilitate the communication that we need to run the various parts of our algorithm. We also present preliminary results on a select set of challenging instances from HyperBench.

This chapter is based on ongoing work in collaboration with Mathias Lanzinger.

5.1 The Candidate Tree Decomposition Framework

The distributed approach we will introduce in this chapter is built on key ideas from Gottlob et al. [39]. We thus present here the underlying definitions.

Vertex-set components. Before we can introduce the definitions, we have to first re-introduce some key notations from Chapter 2, but with a small change to match the notation in [39]. The following framework uses a slightly different definition of *component*, namely components as vertex sets. For a given hypergraph H and a set $W \subseteq V(H)$, a set of vertices $C \subseteq V(H)$ is $[W]$-*connected* if for any two distinct vertices $v, v' \in C$ there exists a sequence of vertices $v_1, \dots, v_h$ and a sequence of edges $e_0, \dots, e_h$ $(h \geq 1)$ with $v_1 = v$ and $v_h = v'$ such that $v_i \in e_{i-1} \cap e_i$ and $v_i \notin W$ for each $i \in \{1, \dots, h\}$. A set $C \subseteq V(H)$ is a $[W]$-*component*, if C is maximal $[W]$-connected. For the remainder of this chapter, the definition of component shall be understood as defined in this paragraph.

Definition 5.1 (Candidate Tree Decomposition [39])**.** Let H be a hypergraph and $\mathcal{T} = \langle T, (B_u)_{u \in T} \rangle$ be a TD of H. Let the candidate bags $\mathbf{S}$ be a family of subsets of $V(H)$. If for each $u \in T$ there exists an $S \in \mathbf{S}$ such that $B_u = S$, then we call $\mathcal{T}$ a *candidate tree decomposition* of $\mathbf{S}$. We denote by $\mathbf{CTD}(\mathbf{S})$ the set of all candidate tree decompositions of H.

Definition 5.2 (Component Normal Form [39])**.** A tree decomposition $\mathcal{T} = \langle T, (B_u)_{u \in T} \rangle$ of a hypergraph H is in component normal form (ComNF) if for each node $r \in T$, and for each child s of r there is *exactly one* $[B_r]$-component C_s such that $V(T_s) = C_s \cup (B_r \cap B_s)$ holds. We say C_s is the component *associated with* node s.

Definition 5.3 ([39])**.** Let H be a hypergraph and let $\mathbf{S}$ be a family of subsets of $V(H)$. Let $\mathcal{T} \in \mathbf{CTD}(\mathbf{S})$ be a TD in ComNF. We say $\mathcal{T}$ is a ComNF candidate tree decomposition of $\mathbf{S}$. We denote by $\mathbf{ComCTD}(\mathbf{S})$ the set of all ComNF candidate tree decompositions of H.

The definitions are given for TDs. However, the algorithm presented in [39] is applicable to output GHDs as well, since the focus in that work is only on the bags that make up a decomposition. The search for the GHD of a given width k of a hypergraph H can also be formulated as the problem of first computing all possible bags with edge cover number k, and then looking for a candidate tree decomposition for this computed set.

The authors in [39] also state a dynamic programming algorithm to show that given a fixed set $\mathbf{S}$, it is polynomial (in fact even in LogCFL [91]) to find a candidate tree decomposition of $\mathbf{S}$. The algorithm that we shall design as part of this chapter will differ in one major way from this algorithm: instead of an eager computation of all bags, we will not need to compute all bags at once and instead lazily compute them as we encounter new subhypergraphs. Before we can define our algorithm, we need a way to encode the idea of a subproblem and its connection to the existence of certain kinds of candidate tree decompositions.

Definition 5.4 (Block [39])**.** Given a hypergraph H, we say that a pair (B, C) of *disjoint* subsets of $V(H)$ is a *block* if C is a $[B]$-component for H or $C = \emptyset$. Such a block is *headed* by B. Let (B, C) and (X, Y) be two blocks. We say that $(X, Y) \leq (B, C)$ if $X \cup Y \subseteq B \cup C$ and $Y \subseteq C$. A block (B, C) is called *satisfied*, if there exists a ComNF TD of $H[B \cup C]$, where the root has bag B. Note that if $C = \emptyset$, then the block is trivially satisfied.

Lemma 5.5 ([39])**.** *Let H be a hypergraph and $B \subseteq V(H)$. If all blocks headed by B are satisfied, then H has a ComNF tree decomposition where B is the bag of the root.*

In the sequel, we will refer to the two elements of a block (B, C) by the name *head* for the first element of the pair B, and *tail* for the second element C. We note that a *satisfied* block (B, C) corresponds to the problem of finding a decomposition $\mathcal{D}$ for a given subhypergraph H, where $V(H) = C$ and the bag of the root node of $\mathcal{D}$ is exactly B. In the algorithm `log-`k`-decomp` from Chapter 4, this is analogous to a single call of the recursive function `Decomp` with the connection subhypergraph $H' = \langle E(H), \emptyset, B) \rangle$ as the input and the function returning “Accept”.

Definition 5.6 (Basis [39])**.** For a block (B, C) and vertex set $B' \subseteq V(H)$ with $B' \neq B$ and let $(B', C_1^{'}), \ldots, (B', C_\ell^{'})$ be all blocks headed by B' such that $(B', C_i^{'}) \leq (B, C)$ for all $i \in [1, \ell]$. We say that B' *is a basis of* $(B, C))$ if the following conditions hold:

(1) $C \subseteq B' \cup \bigcup_{i=1}^{\ell} C_i^{'}$,

(2) for each $e \in E(H)$ such that $e \cap C \neq \emptyset$, $e \subseteq B' \cup \bigcup_{i=1}^{\ell} C_i^{'}$,

(3) and for each $i \in [1, \ell]$, the block $(B', C_i^{'})$ is satisfied.

Lemma 5.7 ([39])**.** *Let (B, C) be a block, and B' a basis of (B, C). Then (B, C) is satisfied.*

Lemma 5.8 ([39])**.** *Let H be a hypergraph and $\langle T, (B_u)_{u \in T} \rangle$ be a ComNF TD of H. Let $r \in T$ be a non-leaf node. For each child s of r, let C_s be the $[B_r]$-component associated with s. The following two statements are true*

- *$(B_s, D) \leq (B_r, C_s)$ if and only if D is either a component associated with a child of s, or if $D = \emptyset$,*
- *B_s is a basis of the block (B_r, C_s).*

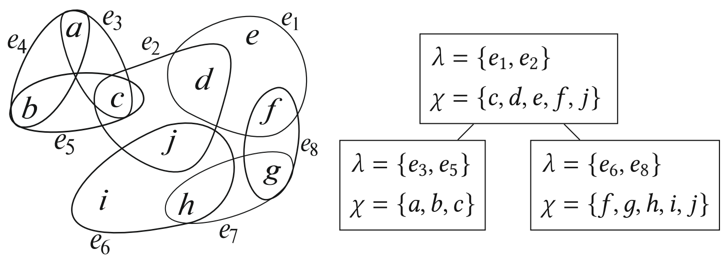
Algorithm 5.1: ComNF Candidate Tree Decomposition

```
Input: A hypergraph $H$ and a set $\mathsf{S} \subseteq 2^{V(H)}$.
Output: "Accept" if ComCTD(S) $\neq \emptyset$, "Reject" otherwise
1  begin
2      $blocks$ := all blocks headed by any $B \in \mathsf{S}$
3      Mark all blocks $(B, C) \in blocks$ as $satisfied$ where $C = \emptyset$
4      repeat
5          foreach $(B, C) \in$ blocks that is not marked as satisfied do
6              foreach $B' \in \mathsf{S} \setminus \{B\}$ do        ▷ Check if there exists a basis $B'$ of $(B, C)$
7                  if $B'$ is a basis of $(B, C)$ then
8                      Mark $(B, C)$ as $satisfied$
9          if For some $B \in \mathsf{S}$, all blocks headed by $B$ are marked as satisfied then
10             return Accept
11     until no new blocks marked
```

A basis is essentially a witness that shows us that a given block can be satisfied, i.e., there exists a decomposition that covers the component in its tail. With these definitions, we can now present the original CTD algorithm from [39].

Algorithm 5.1 first looks at all bags, and marks all those which are trivially satisfied. Next it performs, bottom-up, the search for a basis for all blocks that are not yet satisfied. This is repeated until no new marked nodes can be found. The blocks correspond to all possible bags of nodes that may appear in a candidate tree decomposition. The algorithm starts at the leafs, which are represented by trivial blocks, and extends them upwards. This process finds CTDs which cover larger and larger subhypergraphs, until a CTD for the entire hypergraph is found on line 9. This corresponds to checking whether the case defined in Lemma 5.5 is fulfilled.

Example 5.1. We shall show an example run of Algorithm 5.1. As the input hypergraph, we fix H to be the one visually defined in Figure 5.1. We are also given a set of blocks in the same figure. We shall assume that we are already on line 2, and the set *blocks* is exactly as shown in the table. For the sequel, we shall simply identify each block with its stated ID, as seen in the table. On line 3, the algorithm will mark all trivial blocks as satisfied. In our example these are the three blocks $(\{a, b, c\}, \emptyset)$, $(\{c, d, e, f, i\}, \emptyset)$ and $(\{f, g, h, i, j\}, \emptyset)$. Then we enter the loop which runs from line 4 to line 11. We note that the block 5 has as basis the header $\{f, g, h, i, j\}$. This is due to the trivial block 6, which covers the edges e_6 and e_7, which intersect with the tail of block 5. We also note that block 4 has as basis the header $\{a, b, c\}$, since edges e_3, e_4 and e_5 are covered by the trivial block 1. We find no more unmarked blocks that have a basis, and exit the loop. On line 9, we check if there is a single header such that all its blocks are satisfied. Indeed, the header $\{c, d, e, f, i\}$ has all its blocks satisfied after the first round of the loop. Thus the algorithm accepts.



ID	Block Conent
1	$(\{a, b, c\}, \emptyset)$
2	$(\{a, b, c\}, \{d, e, f, g, h, i, j\})$
3	$(\{c, d, e, f, i\}, \emptyset)$
4	$(\{c, d, e, f, i\}, \{a, b\})$
5	$(\{c, d, e, f, i\}, \{g, h, i\})$
6	$(\{f, g, h, i, j\}, \emptyset)$
7	$(\{f, g, h, i, j\}, \{a, b, c, d, e\})$

Figure 5.1: A hypergraph H, a GHD of H with width 2, and a set of blocks over H, as used in Example 5.1

5.2 Distributed Algorithm

After having introduced the necessary definitions and lemmas in Section 5.1, we now proceed to introduce our distributed algorithm.

Our algorithm has three distinct parts. A coordinator which controls the entire procedure, but does not actually do any computational work itself, the workers which perform the computationally expensive search for balanced separators and the CTD Checker, which performs something similar to Algorithm 5.1, but adapted for our setting where we do not have access to all possible bags right away, but instead receive them in a streaming model. The idea here is to determine as soon as possible when the conditions for finding a decomposition of the input hypergraph are met. This corresponds essentially to a lazy computation of the bags, whereas the original algorithm from Section 5.1 can be seen as an eager computation of all possible bags prior to the search for decompositions. Since it is possible that only a small number of blocks need to be considered to find a decomposition, the lazy approach offers potential speedups of the computation time.

The Coordinator algorithm. The pseudo-code for the coordinator is given in Algorithm 5.2. The coordinator is the only algorithm that actually receives input when it starts, the other parts need to be send their inputs via communication channels. These channels are being set up on line 2. We assume here that the underlying platform allows some kind of reliable one-to-one communication. The coordinator first waits for workers and the CTD Checker to *register* themselves. This happens on lines 4 to 12. Note that, as it is written, the coordinator will start once at least one worker and the (singular) CTD Checker have registered themselves. In case of multiple workers, this means that all workers need to register themselves before the CTD Checker does. Next the coordinator sets up a heap, on line 13. This heap takes subhypergraphs as input. It is called the component heap, as these subhypergraphs will be the output from performing various separations on the input hypergraph, thus the name. The heap is initialised on line 14, with the input hypergraph as the first element. After this begins the main loop of the coordinator, from lines 15 to 37. The coordinator selects the *current* subhypergraph on line 16. In case *current* has less than k edges, the coordinator will skip the search and directly send its

Algorithm 5.2: Coordinator

```
Input: A hypergraph $H$
Parameter: An integer $k \geq 1$.
Output: "Accept" if there is a GHD of width $k$ for $H$, "Reject" otherwise
begin
    inputChannel, toWorker, toCTD := setting up communication channels
    workersRegistered := false
    repeat
        message := ← inputChannel                         ▹ wait for incoming message
        if message is from a worker then
            workerID := numWorkers + 1
            numWorkers := numWorkers + 1
            toWorker ← confirm registration of Worker and send workerID and $H$
        else message from CTD Checker
            toCTD ← confirm registration of CTD Checker and send $H$
    until at least one Worker and the CTD Checker registered
    compHeap := initialise empty component heap
    compHeap.Add ( $H$ )                    ▹ initialise the heap with the input hypergraph
    repeat
        current := compHeap.getNextElement()
        if |current| ≤ $k$ then
            toCTD := send trivial blocks to CTD Checker
        else
            generators := create |numWorkers|-many generators with edges from current
            toWorkers ← generators and current     ▹ each worker gets a full copy of generators
        allWorkersIdle := false
        numIdle := 0
        repeat
            message := ← inputChannel                     ▹ wait for incoming message
            if message is from worker and confirms that it's idle then
                numIdle := numIdle + 1
                if |numIdle| = |numWorkers| then
                    allWorkersIdle := true
            else if message is from worker and reports new subhypergraphs then
                newSHGs := extract new subhypergraphs from message
                compHeap.Add ( newSHGs )          ▹ keep track of new found subhypergraphs
            else if message from CTD Checker and confirms that a decomp found then
                toWorker ← tell workers to end the search
                return Accept
        until allWorkersIdle
    until compHeap is empty
    toWorker ← tell workers to end the search
    return Reject
```

Algorithm 5.3: Worker

```
1  begin
2      $inputChannel, toCoord, toCTD$ := setting up communication channels
3      $toCoord$ ⟵ send registration request to Coordinator
4      $workerID$ := ⟵ $inputChannel$                  ▷ wait for response and set worker id
5      $H$ := ⟵ $inputChannel$                ▷ receive input hypergraph from coordinator
6      $searchActive$ := true
7      repeat
8          $message$ := ⟵ $inputChannel$                         ▷ wait for incoming message
9          if  message from Coordinator; ending the search then
10             $searchActive$ := false
11         else if  message from Coordinator; sending new generators then
12             $current$ := extract from message
13             $generator$ := extract from message, using workerID as index
14             foreach $sep \in generator$ do
15                 if $sep$ is a balanced separator of $current$ then
16                     $blocks$ := create all blocks $(B, C)$ where $B \subseteq \bigcup sep$ and $C$ a
                         $[B]$-component of $H$
17                     $toCTD$ ⟵ $blocks$                  ▷ sending blocks to CTD Checker
18                     $comps$ := $[sep]$-components of $current$
19                     $toCoord$ ⟵ $comps$         ▷ sending new components to Coordinator
20             $toCoord$ ⟵ sending idle message to Coordinator
21     until not $searchActive$
```

blocks to the CTD Checker. Otherwise it generates the *generators* on line 20. These generators iterate over all choices of k edges among the subhypergraph *current*. The pair of *generators* and *current* is send to all workers. Finally, the coordinator waits for a response in the following loop, running from line 24 to 36. It waits for a message, and checks its contents. If the message is from a worker signalling that it is idle (i.e. it has finished its slice of the search), then the coordinator checks if all workers are idle on line 28, if so, the coordinator sets a flag and will continue with a new subhypergraph. If a worker has produced new subhypergraphs, then the coordinator will add these to the heap on line 32. Lastly, if the CTD Checker reports that it has found a decomposition, the coordinator signals the workers to quit and returns “Accept”. If the heap has been fully exhausted and no decomposition has been found, then the coordinator tells the workers to quit on line 38 and returns “Reject”.

The Worker algorithm. Unlike the coordinator, the worker receives no input when it starts. Instead it first sets up the required communication channels on line 2. Next it *registers* itself with the coordinator. Note that this design choice means that the coordinator has to be running first, thus inducing an order on which parts of our distributed algorithm need to be started first. Once the worker receives the input hypergraph and its workerID from the coordinator on

Algorithm 5.4: CTD Checker

```
1  begin
2      $inputChannel, toCoord$ := setting up communication channels
3      $toCoord \leftarrow$ send registration request to Coordinator
4      $H := \leftarrow inputChannel$                        ▷ receive input hypergraph from coordinator
5      $decompFound$ := false
6      $rootBlock := (\emptyset, V(H))$
7      $blocks := \{rootBlock\}$                     ▷ Add the root block as the starting point
8      repeat
9          $message := \leftarrow inputChannel$                       ▷ wait for incoming message
10         if $message from Coordinator; ending the search$ then
11             break
12         else if $message from Worker; delivering new blocks$ then
13             $newBlocks$ := extract from message
14             $blocks \leftarrow blocks \cup newBlocks$
15             foreach $(B, C) \in$ blocks do
16                 foreach $(B', C') \in$ blocks *where* $(B', C') \neq (B, C)$ do
17                     if $B'$ *is a basis of* $(B, C)$ then
18                         Mark $(B, C)$ as satisfied
19             if *rootBlock is marked as satisfied* then
20                 $toCoordinator \leftarrow$ send message that decomposition was found
21                 $decompFound$ := true
22     until *not* $decompFound$
```

lines 4 and 5, it starts the main loop. This loop runs from lines 7 to 21. It begins by waiting for a message. If the message is from the coordinator and signals an end to the search, then the worker sets a flag and will subsequently terminate. Otherwise, it receives from the coordinator a set of generators and a subhypergraph *current*. On line 13, the worker uses its own workerID as the index to select its own generator from the entire set it received. The workerID only serves the purpose of being an index for this set. Next the worker iterates over all edge combinations produced by the generator, seen on lines 14 to 19. It checks if the current edge set is a balanced separator for *current*. If so, it generates all its blocks. Note that these are based on components generated by separating the global input hypergraph H. These blocks are sent to the CTD Checker on line 17. Next the worker produces the components when separating against *current*, on line 18 and sends these to the coordinator on line 19. This allows the coordinator to direct the search for all relevant blocks. Once the worker is done with its current search, it sends an idle message to the coordinator on line 20.

The CTD Checker algorithm. The CTD Checker receives no input when it starts. On line 2, it sets up the needed communication channels. Next it sends a registration request to the coordinator on line 3. It receives as answer the input hypergraph H. Next it sets up the *rootBlock* on line 6. This will play a role when determining whether a decomposition has been found. It uses the empty set as the head, and the entire input hypergraph as its tail. Note that this is using vertex sets as components, to follow the definitions. Next the set of all currently known blocks is set up on line 7, with *rootBlock* as the first block. After this, the CTD Checker enters its main loop which runs from lines 8 to line 22. It waits for a new message on line 9. If the coordinator signals that all subhypergraphs have been searched to exhaustion, then the CTD Checker breaks from the loop and subsequently terminates. Otherwise, it can receive a new block from one of the workers. It adds the newly found blocks to the known set on line 14. After this, it performs a comparison among all known blocks on lines 15 to 18, checking which blocks are currently satisfied. After this, on line 19 the CTD Checker tries to see if *rootBlock* is satisfied. If so, it reports to the Coordinator on line 20 that a decomposition has been found. It sets a flag and subsequently terminates.

Example 5.2. We will now go through an example run of the distributed algorithm. As with Example 5.1, we shall use the hypergraph H from Figure 1.2 as the input. The parameter for k is set to be 2, and we assume an instance of the Coordinator, a number of instances of the Worker and one instance of the CTD Checker are already running and have set up their respective communication channels. The Coordinator begins by registering all instances of the workers and the sole CTD Checker instances, as seen from lines 6 to 16. Next the *compHeap* is initialised with the input hypergraph H on line 18. The Coordinator proceeds to send the current subhypergraph from the heap to the workers on lines 24 to 25, unless a trivial case is found where the current subhypergraph has size k or less, in which case the Coordinator skips the workers and directly sends the trivial blocks to the CTD Checker in their stead. Assuming the workers did receive some input, the Coordinator waits for them finish. Meanwhile, the workers look for all balanced separators of H. At width 2, there are 7 such balanced separators. The reason is that the edge e_2 itself is already a balanced separator, and thus every combination of it with the remaining 7 edges is one too. For each such combinations, the workers will send the produced components to the Coordinator, which in turn sends these back to the workers for more balanced separators to find, until only graphs of size 2 or less remain. We will not try to list all possible balanced separators here. Once enough blocks have reached the CTD Checker, it will find at least one block that in turn satisfied the *rootBlock*. We will not go into detail here, as the this corresponds essentially to Algorithm 5.1 and we already have an example run of it in Example 5.1. We assume here that CTD Checker will eventually reach line 20 and tell the Coordinator to end the search. The Coordinator in turn sends a message to the workers to end their search on line 38 and then returns Accept on the next line.

5.2.1 Correctness of the Distributed Algorithm

Next, we will establish the correctness of the distributed algorithm. Our strategy for this will be to show a translation of successful runs of Algorithm 5.1 into runs of our distributed algorithm, for a specified set of bags. Before we can do this, however, we first need to establish the set of bags of nodes which are computed during the run of the distributed algorithm for a given hypergraph and width parameter. Our goal is to show that this set is the same set of bags as are considered by the Parallel Balanced Separator algorithm given in Chapter 3, defined in Algorithm 3.1. That these two sets coincide is of course no coincidence, we specifically designed our distributed algorithm with this property in mind. The reason we chose as our target set of bags the ones produced by the Parallel Balanced Separator algorithm from Chapter 3 is that we want to utilise the Balanced Separators Approach. To formally prove our claim, we first need to define the set of bags in question.

To speak about all possible bags produced by Algorithm 3.1, we need to define the possible subhypergraphs visited by the algorithm. We define a few helper functions to make this easier.

$$\begin{aligned}\texttt{BalancedComp}(H',k) &= \{c \mid \exists sep \in E(H')^k \text{ s.t. } sep \text{ is a balanced separator of } H' \\ &\qquad \wedge\, c \text{ is a } [sep]\text{-component of } H'\} \\ \texttt{ParBalSepHGs}(H',k) &= H' \cup \bigcup_{c \in \texttt{BalancedComp}(H',k)} \texttt{ParBalSepHGs}(c,k)\end{aligned}$$

The function $\texttt{ParBalSepHGs}(H',k)$ describes the set of subhypergraphs visited by Algorithm 3.1 from Chapter 3, when given as input the hypergraph H' and using the integer k for the width parameter. Due to the recursive nature of the algorithm it describes, this function too is recursive.

Lemma 5.9. *For a given hypergraph H and a positive integer k, the set of all possible values assigned to subSep on line 11 of the Parallel Balanced Separator algorithm, given in Algorithm 3.1 in Chapter 3, is exactly*

$$\begin{aligned}\textit{BagsGHD}(H,k) = \{X \mid \exists \mathrm{sep} \in E(H)^k \ \textit{s.t.}\ X \subseteq \mathrm{sep} \wedge \\ \exists C \in \textit{ParBalSepHGs}(H,k)\ \textit{s.t.}\ \mathrm{sep}\ \textit{is a balanced separator of}\ C\}\end{aligned}$$

Proof of Lemma 5.9. To show this claim, we first observe that the function $\texttt{ParBalSepHGs}(H,k)$ captures exactly the subhypergraphs which are traversed during a run of Algorithm 3.1: Of course H itself is included and if there exist balanced separators of size k for H, then the algorithm continues recursively by traversing any components created by separating H against these balanced separators. This ends until either a subhypergraph is found that permits no balanced separators of this size, or until we encounter the trivial case where the currently investigated subhypergraph itself has edge cover number of k or less. Algorithm 3.1 assigns to subSep vertex sets based based on its choice of edge cover in *sep*. These are the balanced separators it is trying to find. In addition to the full set of vertices in *sep*, subSep can also be assigned every possible subset of *sep*, as seen on line 20. Of course, this is exactly what is expressed in the set $\texttt{BagsGHD}(H,k)$. □

Next we establish that our distributed algorithm is designed to consider blocks with heads that match exactly this same set.

Lemma 5.10. *For a given hypergraph H and a positive integer k, the union of all heads of blocks produced by the Worker instances on line 16 of Algorithm 5.3 during a run of the distributed algorithm is exactly the same set as* $\texttt{BagsGHD}(H, k)$.

Proof of Lemma 5.10. This claims follows from the observation that the set of all subhypergraphs investigated by our distributed algorithm, for a given input hypergraph H and a positive integer k, is exactly $\texttt{ParBalSepHGs}(H, k)$. To see that this observation holds, we look at how the distributed algorithm chooses new subhypergraphs for the workers to investigate. This is controlled by the Coordinator, described in Algorithm 5.2. The Coordinator uses the *compHeap* to determine which subhypergraph to send to the workers next. This set is initialised by H, as can be seen on line 18. After this, the Coordinator waits for workers to send it new components, as seen on lines 35 to 36. The workers, on the other hand, investigate one subhypergraph at a time, and try to find balanced separator of size k from the set $E(H)$ such that the separator splits the *current* subhypergraph into components at most half the size of the subhypergraph itself. If no such separator exists, they do nothing. If they do find a balanced separator *sep*, they compute all relevant blocks, seen on line 16 of Algorithm 5.3 and they also compute $[sep]$-components of *current*, where *current* is the subhypergraph it is currently investigating. Then the workers sends these components back to the Coordinator on line 19. We can observe that this reflects exactly the construction of $\texttt{ParBalSepHGs}(H, k)$. The only thing left to argue is the nature of the specific bags created. These bags are exactly the *heads* of the blocks that the worker creates. As was mentioned, this happens on line 16 of Algorithm 5.3. We see that the worker indeed iterates over all possible subsets of the balanced separator. Thus, this corresponds exactly to the set $\texttt{BagsGHD}(H, k)$. □

At this point, we can already make use of the lemmas from Section 5.1 to make some observation about the set of bags $\texttt{BagsGHD}(H, k)$, for a given hypergraph H and positive integer k. By the correctness of the Parallel Balanced Separator algorithm, given in Algorithm 3.1in Chapter 3, we know that it accepts if and only if there is a GHD of H with width k or less. More strictly, the Parallel Balanced Separator algorithm searches for GHDs which are in a normal form which is stricter than ComNF. Thus, by Lemma 5.9, if it accepts on input H and k, we have that $\mathbf{ComCTD}(\texttt{BagsGHD}(H, k)) \neq \emptyset$. Putting all this together, we get the following corollary.

Corollary 5.11. *For a given hypergraph H and positive integer k, it is the case that there is a GHD for H of width k or less if and only if* $\mathbf{ComCTD}(\texttt{BagsGHD}(H, k)) \neq \emptyset$.

Now we have all we need to establish the correctness of the distributed algorithm we have introduced in this section.

Theorem 5.12. *The distributed algorithm described here consisting of three parts running concurrently on separate machines: a single instance of the coordinator, described in Algorithm 5.2, a single instance of the CTD Checker as described in Algorithm 5.4 and some number of instances of the worker as described in Algorithm 5.3, will return accept at the coordinator instance when given as input a hypergraph H and a positive integer k if and only if* $\mathbf{ComCTD}(\texttt{BagsGHD}(H, k)) \neq \emptyset$.

Proof of Theorem 5.12. By Lemma 5.10, we know that our distributed algorithm will eventually produce the entire set $\texttt{BagsGHD}(H, k)$ as blocks, which are pairs (B, C) where B is an element of $\texttt{BagsGHD}(H, k)$ and C is a $[B]$-component of H. These blocks will eventually all arrive at the CTD Checker, as described in Algorithm 5.4. Each time a new block arrives, the CTD Checker repeats the lines 15 to line 21. Note that the nested loops from line 15 to line 18 – used to mark which blocks are already satisfied – is exactly the same as on lines 5 to 8 from Algorithm 5.1. Next the CTD Checker will try to see if the root block has been satisfied on line 19 of Algorithm 5.4. Note that the only way the root block $(\emptyset, V(H))$ could be satisfied, is if there is a basis B' s.t. all blocks headed by B' are marked as satisfied. The reason for this is that the tail of the root block is the entire input hypergraph, and thus a basis of it needs to cover every edge in the input hypergraph, in every $[B']$-component of H. This description must include all possible blocks that can be headed by B'. Therefore, this is the same condition as the check on line 9 of Algorithm 5.1. In other words, the CTD Checker – once it received every block produced by the workers – will accept if and only if Algorithm 5.1 accepts when it is given as input the set of bags $\texttt{BagsGHD}(H, k)$. As was shown in [39], Algorithm 5.1 is sound and complete, and thus so is our distributed algorithm. □

5.3 Optimising the Basis Check

In this section we will focus on a key challenge for an efficient implementation of the distributed algorithm we introduced in Section 5.2. Since the original algorithm from [39] was not meant to be implemented as is, it simply described the search for a basis of a block as a nested loop search among all possible pairs of blocks, and this search is repeated until a fixpoint is reached. This means we need at least a quadratic number of comparisons among all possible bags. If we translate this approach to our distributed setting, where we compute the set of bags in a lazy fashion, it means that the check for which blocks are satisfied by searching for a basis will decrease in performance by a square function of the number of all bags found so far. While this is not major problem for small hypergraphs and widths, we shall remind the reader that our entire reason for this line of research was to solve instances that all other approaches failed to optimally solve so far, and these include some of the largest instances of HyperBench with over 100 hyperedges. If we consider these large instances, then iterating through all pairs of blocks becomes very unrealistic, even for relatively low widths such as 5. At 100^5, it would already be challenging to keep that set in memory and the quadratic time needed for the basis check would be a major obstacle to solving such instances in reasonable time.

Thus we set out to find ways to optimise the search for a basis for a given block among a set of already computed blocks, or the "basis check", as we shall call it in this section. In order to find ways to speed-up the basis check, we need to find properties which every basis needs to fulfil and which allow us to quickly discard large parts of the search space. The definition of a basis is given in Definition 5.6. For a given block (B, C), a basis B' is a vertex set such that there exists a set of blocks $(B', C'_1), \dots, (B', C'_\ell)$ such that $(B', C_i) \le (B, C)$ for all $i \in [1, \ell]$. The definition of the "$\le$" operator for two blocks is given in Definition 5.4. For two blocks (B, C), (X, Y), we have that $(B, C) \le (X, Y)$ iff $B \cup C \subseteq X \cup Y$ and $C \subseteq Y$. Thus, for one block to be smaller than another, its tail has to be a (not necessarily strict) subset of the other block's tail. We will use

Table 5.1: A visual example for the look-up table used to find supersets quickly.

Element	Sets containing it
a_0	$\{S' \mid S' \in \mathcal{S} \land a_0 \in S'\}$
$\vdots$	$\vdots$
a_i	$\{S' \mid S' \in \mathcal{S} \land a_i \in S'\}$
$\vdots$	$\vdots$
a_n	$\{S' \mid S' \in \mathcal{S} \land a_n \in S'\}$

this property to pre-filter the blocks which need to be considered for the basis check: Given a set of blocks $\mathcal{B}$ and a particular block $(B, C) \in \mathcal{B}$, we want to quickly determine the following sets:

$$\text{SubsetBlocks}(\mathcal{B}, (B, C)) = \{(B', C') \mid (B', C') \in \mathcal{B} \land C' \subseteq C\}$$
$$\text{SupersetBlocks}(\mathcal{B}, (B, C)) = \{(B', C') \mid (B', C') \in \mathcal{B} \land C \subseteq C'\}$$

The reason we need both the subsets and superset blocks is that we want to avoid having to run the entire nested search every time a new block arrives. Instead, the goal is to determine two minimal sets of “relevant” blocks: those blocks which might be part of a basis of the newly added block, and vice-versa, those blocks for whom the new block itself might act as (part of) a potential basis. By using the knowledge that only these sets need to be considered, we avoid the need to look at al possible pairs of blocks every time a new block arrives.

In order to find these sets efficiently, we have implemented a custom data structure for the superset check. As we will explain, this same data structure can actually be used to look for subsets as well, though for this translation of the search for subsets to the search for supersets we will need to fix the domain or “universe” among the family of sets we get as inputs. This universe is simply the set $V(H)$ for the given input hypergraph H.

We will also introduce a look-up table, which for any given element in the universe $v \in \mathcal{U}$, and a family of subsets $\mathcal{S} = \{S_0, \ldots, S_n\}$ where $S_i \subseteq \mathcal{U}$ for each $i \in [0, n]$, will produce the set $\{S' \mid S' \in \mathcal{S} \land v \in S'\}$. The table is internally implemented as an associated array, or map, and allows constant time access to these sets. The table itself can be generated in a preprocessing step in linear time: simply traverse each element in a set, and indicate for each element in the table which additional index needs to be added to its value list.

We now proceed to introduce the two data structures. For each, we have two methods. One for indexing a new set, updating the internal look-up table and another method which takes as input an arbitrary set and outputs all index sets which are subsets (resp. supersets) of the input set. We assume in the given pseudo-code that each set has a unique identifier by which it can be referred to, and thus it is not necessary to always copy the entire set. This identifier could be as simple as an index in a global list of all unique sets encountered so far.

Algorithm 5.5: Superset Search data structure

```
   Type: $LookUp$ = a mapping from $V(H)$ to $2^E(H)$
   Type: $allSets$ = list of all sets indexed so far
1  Method AddSet($inputSet$)
2      foreach $v \in$ inputSet do
3          $currentSets := LookUp[v]$                          ▷ Check current mappings for $v$
4          $currentSets = currentSets \cup inputSet$
5          $LookUp[v] = currentSets$                           ▷ Adding inputSet to value list
6      $allSets = allSets \cup inputSet$
7  Method GetSuperset($inputSet$)
8      if $inputSet = \emptyset$ then
9          return $allSets$                                    ▷ Special case to handle $\emptyset$
10     $hashTable$ := initialise a hash table
11     foreach $v \in$ inputSet do
12         $currentSets := LookUp[v]$                          ▷ Check current mappings for $v$
13         foreach $s \in currentSets$ do
14             if $mapping$ hashTable$[s]$ $does\ not\ exist$ then
15                 $hashTable[s] = 1$                          ▷ first time this indexed set was encountered
16             else
17                 $hashTable[s] = hashTable[s] + 1$           ▷ increment the value
18     output := $\emptyset$
19     foreach $key\ value\ s \in hashTable$ do
20         if $hashTable[s] = |\ inputSet\ |$ then
21             output = output $\cup\ s$
22     return output
23 Method GetAllSets()
24     return $allSets$
```

Superset data structure. The superset data structure is detailed in Algorithm 5.5. The way this should be read is that there is a single object, which has two methods by which it can be accessed. The first is `AddSet`, which takes as input a subset of $V(H)$ for some hypergraph H (in fact, there is no need to explicitly provide the entire set $V(H)$). `AddSet` iterates over all vertices of this input set and updates the look-up table to know which vertices occur in which indexed set. The other method `GetSupersets` also takes an inputSet as argument and then does the following. The goal is to look at all sets that share some element of the input set, and to compute the *intersection* of them. In order to do this efficiently, we need slightly more complicated construction, however. We use a hash table, defined on line 10. This will map each set to a count of vertices of the input set which it contains. So for each vertex of the input set, we use the look-up table to get the indexed sets which share this vertex. Then we go over this

Algorithm 5.6: Subset Search data structure

```
Type: superSet = superset data structure; detailed in Algorithm 5.5
Method AddSet(inputSet, universe)
    diffSet := universe \ inputSet
    superSet.AddSet(diffSet)
Method GetSubset(inputSet, universe)
    diffSet := universe \ inputSet
    if diffSet = ∅ then
        temp := superSet.GetAllSets()                              ▷ Special case
        output := ∅
        foreach s ∈ temp do
            s = universe \ s
            output = output ∪ s
        return output
    superSetsDiffs := superSet.GetSuperset(diffSet)
    output := ∅
    foreach s ∈ superSetsDiffs do
        s = universe \ s
        output = output ∪ s
    return output
```

set of indexed sets, and increment the corresponding value in the hash table, or initialise it to 1 if it is the first time we encounter this set. Intuitively, the hash table just tells us how large the intersection with the input set is, for a given indexed set. Clearly, for any superset, this intersection must be the size of *inputSet* itself. Thus, on lines 19 to 21, we traverse the hash table, and only output the index sets which meet this condition. Note that the empty set requires special treatment in `GetSuperset`, for this purpose we keep a separate list of all indexed sets and return it when we get as input the empty set.

Subset data structure. The subset data structure is detailed in Algorithm 5.6. The basic observation here is this: given a universe $\mathcal{U}$ and subsets $S_1, S_2 \subseteq \mathcal{U}$, then we have that $S_1 \subseteq S_2$ if and only if $\mathcal{U} \setminus S_2 \subseteq \mathcal{U} \setminus S_1$. Thus, if we fix the universe against which to perform the set difference, we can express the subset relation via the superset relation of the inverse sets, where *inverse* is understood as the set difference against the universe. As can be seen in the algorithm, the two methods `AddSet` and `GetSubsets` require the universe as an additional argument, in addition to an input set. Each method call then translates the input to the inverse against the universe and calls the superset data structure defined above. As before, there is a need for a special case. For the subset data structure, we need to consider the case that the input set is exactly the universe. In this case we return all sets from the superset data structure, after inverting them again.

5.4 Implementation and Preliminary Experimental Evaluation

In this section, we want to present experimental data on the current prototype that we have developed as part of our ongoing work. We stress that this is our first working version, and essentially still a proof of concept. As such there is still ample room for improvement. We have implemented the distributed algorithm we described in Section 5.2. Our implementation already incorporates the optimisation to the base check we detailed in Section 5.3. In this section, we also report on some preliminary results of our prototype. We have already some very promising results and show in select cases clear improvements over our previous decomposition methods.

5.4.1 Prototype Implementation

We have chosen the Google Cloud Platform (GCP) as the platform for our prototype. We stress that the algorithm we presented in this chapter is not limited to the GCP and can be implemented on any distributed system. The only requirement is that some form of rudimentary message passing between individual machines needs to be supported, which is a requirement that – to our knowledge – is met by every commercial cloud platform. One reason for choosing GCP was its good integration with the Go programming language, with many of its APIs being implemented as Go libraries. We chose the Go programming language in order to quickly build on the shared code base between BalancedGo and `log-k-decomp`. We wanted to use the same idea of communication channels from Go in the distributed setting as well. Thus we decided to make use of Google PubSub, a messaging middleware. In PubSub, messages are sent to "topics" and can be read via individual subscriptions to these topics, where each subscription gets distinct copies of each message sent to a topic. For our implementation, we need only three such topics: "to_worker", "to_coordinator" and "to_CTDCheck". For the workers, this means that in principle all workers get the same messages. Using their unique ID, they can then extract the part of the message block unique to them. As there is only one unique coordinator and one CTDChecker instance active during any run, the recipient in those two cases is always clear.

5.4.2 Methodology

For our experiments, we used a VM on the GCP, running on the following software and hardware. The VM was running the Linux distribution Debian 10, using kernel version 4.10.020. The underlying CPU was the AMD EPYC 7B13, running 112 logical cores, clocked at 2.5 GHz and the available RAM was 224 GB. For these experiments, we ran all instances of the distributed algorithm on one machine. This restriction allowed us to better compare our distributed algorithm against existing methods, as we can simply run them on the exact same hardware. We do note, however, that the distributed algorithm is designed to run on an arbitrary amount of machines. We proceed to detail how we ran our experiments.

The Need for Testing on a Small Sample. As the underlying benchmark, we will once again use HyperBench [25]. We are not interested on running experiments on all of HyperBench, however. Instead, we want to show how effective our prototype is at solving what we call "hard instances". There are two reasons for this. The first is of practical interest. Each test run on

Table 5.2: Groups of instances from HyperBench which could not be solved by `log-`k`-decomp`, as presented in Chapter 4. The groups are based on instances with the same origin, as defined in [25].

Group names	Number of instances
S	16
Kakuro	43
Nonogram	159
Pi	58
aim	20
cnf	24
reg	100
rand	106
Other	20
Total	546

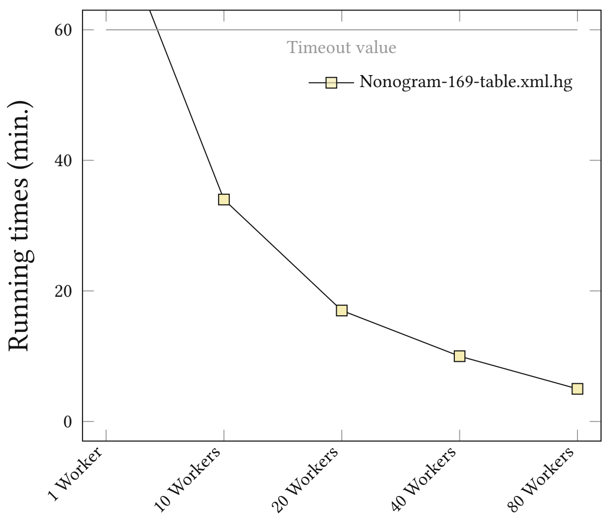


Figure 5.2: Experiment showing the runtime with different number of worker processes involved for a single instance from `hardInstances`. A timeout of 60 minutes was used, as indicated.

the Google Cloud Platform over our chosen benchmark instances will generate a certain cost. This is contrast to our experiments in Chapter 3 and Chapter 4, where we had access to our own local test servers without per-use costs associated. Since the course of our experiments will require multiple iterations, to ensure the quality and validity of our produced data, this becomes prohibitive in terms of costs if we chose the entirety of HyperBench, or even a very large subset of it. Thus we need to find a small subset, which still allows us to draw meaningful conclusions. The second reason why one would want to restrict the experiments to a subset of HyperBench in the first place is related to the aim of this line of research. Let us recall our

Table 5.3: Overview over elements of `hardInstances`. Pairs of Instances of same origin are grouped together.

Name	# Hyperedges	# Vertices	Unsolved width
s5378.hg	2958	2993	3
s1494.hg	653	661	3
s820.hg	294	312	4
s832.hg	292	310	4
reg-s20-p03-c20-d10-n10-l5-71.xml.hg	100	130	6
reg-s20-p03-c20-d10-n10-l5-06.xml.hg	100	130	6
reg-s20-p03-c20-d10-n10-l5-85.xml.hg	100	130	6
reg-s20-p03-c20-d10-n10-l5-24.xml.hg	100	130	6
rand_q0349.hg	43	92	8
rand-3-28-28-93-632-18.xml.hg	93	28	6
rand-3-28-28-93-632-22.xml.hg	93	28	6
rand_q0472.hg	42	99	10
cnf-3-80-0100-735542.xml.hg	100	79	6
cnf-2-40-0100-730625.xml.hg	96	40	6
cnf-3-40-0100-730621.xml.hg	99	40	6
cnf-3-40-0100-730630.xml.hg	97	40	6
aim-50-2-0-unsat-4.xml.hg	95	50	6
aim-50-1-6-sat-2.xml.hg	76	50	7
aim-50-1-6-sat-4.xml.hg	77	50	7
aim-50-1-6-unsat-2.xml.hg	77	50	7
Pi-40-10-07948-40-71.xml.hg	98	40	6
Pi-40-10-07948-40-18.xml.hg	98	40	6
Pi-40-10-07948-40-12.xml.hg	98	40	6
Pi-40-10-07948-40-90.xml.hg	98	40	6
Nonogram-169-table.xml.hg	48	576	7
Nonogram-009-table.xml.hg	64	1015	6
Nonogram-020-table.xml.hg	64	1024	7
Nonogram-085-table.xml.hg	48	576	8
Kakuro-medium-153-ext.xml.hg	98	169	6
Kakuro-hard-113-ext.xml.hg	90	153	6
Kakuro-easy-103-ext.xml.hg	96	162	6
Kakuro-easy-113-ext.xml.hg	90	153	6
grid2d_50.hg	1250	1250	3
pigeonsPlus-11-05.xml.hg	77	66	5
2bitcomp_5.hg	310	95	4
flat30-99.hg	300	90	5

motivation for this line of research: solving those remaining cases where all prior methods fail to produce optimal decompositions. Hence, we are not primarily interested in providing methods that are good with small or intermediate instances, as this is, at this point, essentially a solved problem. Looking at the detailed raw data from our two previous experiments in Chapter 3 and Chapter 4 [36, 48], we have identified just over 500 instances which could not

Table 5.4: Overview over experiments over `hardInstances`, with a one hour timeout and using 80 workers for the distributed algorithm.

Instance	Distributed	BalancedGo [49]	`log-`k`-decomp` [37]
s5378.hg	timeout	timeout	timeout
s1494.hg	6m 56s	timeout	timeout
s820.hg	23m 50s	timeout	timeout
s832.hg	23m 13s	timeout	timeout
reg-s20-p03-c20-d10-n10-l5-71.xml.hg	27m 47s	timeout	timeout
reg-s20-p03-c20-d10-n10-l5-06.xml.hg	27m 37s	timeout	timeout
reg-s20-p03-c20-d10-n10-l5-85.xml.hg	33m 17s	timeout	timeout
reg-s20-p03-c20-d10-n10-l5-24.xml.hg	33m 53s	timeout	timeout
rand_q0349.hg	timeout	timeout	timeout
rand-3-28-28-93-632-18.xml.hg	timeout	timeout	timeout
rand-3-28-28-93-632-22.xml.hg	timeout	timeout	timeout
rand_q0472.hg	timeout	timeout	timeout
cnf-3-80-0100-735542.xml.hg	22m 27s	timeout	timeout
cnf-2-40-0100-730625.xml.hg	timeout	timeout	timeout
cnf-3-40-0100-730621.xml.hg	20m 55s	timeout	timeout
cnf-3-40-0100-730630.xml.hg	18m 35s	timeout	timeout
aim-50-2-0-unsat-4.xml.hg	13m 56s	timeout	timeout
aim-50-1-6-sat-2.xml.hg	timeout	timeout	timeout
aim-50-1-6-sat-4.xml.hg	41m 35s	timeout	timeout
aim-50-1-6-unsat-2.xml.hg	timeout	timeout	timeout
Pi-40-10-07948-40-71.xml.hg	13m 55s	timeout	timeout
Pi-40-10-07948-40-18.xml.hg	14m 07s	timeout	timeout
Pi-40-10-07948-40-12.xml.hg	timeout	timeout	timeout
Pi-40-10-07948-40-90.xml.hg	timeout	timeout	timeout
Nonogram-169-table.xml.hg	6m 19s	timeout	timeout
Nonogram-009-table.xml.hg	11m 22s	timeout	timeout
Nonogram-020-table.xml.hg	timeout	timeout	timeout
Nonogram-085-table.xml.hg	40m 11s	timeout	timeout
Kakuro-medium-153-ext.xml.hg	25m 21s	timeout	timeout
Kakuro-hard-113-ext.xml.hg	timeout	timeout	timeout
Kakuro-easy-103-ext.xml.hg	timeout	timeout	timeout
Kakuro-easy-113-ext.xml.hg	timeout	timeout	timeout
grid2d_50.hg	timeout	timeout	timeout
pigeonsPlus-11-05.xml.hg	timeout	timeout	timeout
2bitcomp_5.hg	19m 27s	timeout	timeout
flat30-99.hg	timeout	timeout	timeout

be optimally solved by either `log-`k`-decomp` or BalancedGo. We give an overview over these challenging instances in Table 5.2. Thus we want to choose a representative sample among these. Simply taking random elements of this set would not be a good idea, however. As we can see in Table 5.2, these challenging instances fall into 8 groups, of the same origin, with only 20 instances in the remaining “Other” group. For more details on the breakdown of the origins of hypergraphs from HyperBench, we refer to [25]. In our observation, instances of the same origin tend to share strong structural similarities. A truly representative sample should pick equally among each of these groups. We have therefore randomly chosen 4 instances among these 8+1 groups, leading to 36 hypergraphs which we shall identify by the name `hardInstances`. For each element of `hardInstances` we have also identified the smallest positive integer k such that the CheckGHD (resp. CheckHD) in BalancedGo (resp. `log-`k`-decomp`) failed to terminate. The names of the sample instances as well as information on the number of hyperedges and number of vertices can be seen in Table 5.3. The pair of elements of `hardInstances` and this k will make up our set of experiments, where we try to answer the CheckGHD problem using our distributed algorithm.

We set a one hour timeout for all the experiments we will report in this section.

5.4.3 Experimental Results

We see the results of our experiments over the instance set `hardInstances` in Table 5.4. For each instance, we list the observed run times for three methods. The first is our prototype implementation of the distributed algorithm we presented in this chapter, called simply Distributed. Next we have the run times of BalancedGo, the decomposition method introduced in Chapter 3 and the final decomposition method we compare is `log-`k`-decomp`. This implementation was introduced in Chapter 4.

As we claimed, we can see that the two methods BalancedGo and `log-`k`-decomp` fail to terminate for any instance under the one hour timeout, whereas our distributed algorithm managed to terminate and answer the CheckGHD problem in 19 out of 36 cases. We note here that all three methods ran on the same machine here, as detailed in Section 5.4.2. Thus, our distributed prototype is able to far better utilise the same hardware, and the other parallel algorithms seem to be unable to truly scale to use the 112 cores well. What makes this result particularly surprising, is the observation that even in cases where there is not a single balanced separator of the sought after size, such as in two of the four Nonogram cases, the non-distributed algorithm still fail to terminate, whereas the distributed algorithm finishes in up to 6 minutes or just over 11 minutes. As there is large amount of shared code between all three implementations here, particularly in regards to how the search space is split among the number of workers, the key difference must lie in the inherent overhead that comes with having to run a large number of goroutines, which the distributed algorithm avoids by relying on completely separate processes.

In order to investigate how well the prototype scales with an increasing number of workers involved, we report another set of experiments. We took one instance from `hardInstances` and we wanted to see how the runtime is affected by using more and more worker instances. We can see the results in Figure 5.2. For a single worker, the run did not terminate under one hour. From 10 workers onward, we can see fairly good improvements up to 80 workers. We also observe, however, that the scaling is not quite linear, but that is to be expected as we reach running times in the single-digit minutes. The distributed algorithm requires the coordination of a large number of instances, and has an inherent communication delay. In addition to this, it also needs some time to fully start up and to terminate all running instances when it reached the end. Of course, this is still a strong improvement over the existing methods, which fail to solve this particular instance in one hour.

5.5 Summary

We have designed a novel distributed algorithm for computing hypergraph decompositions and shown its correctness for solving the CheckGHD problem. While the exact version introduced here is for computing GHDs, it would only require a different definition of the Worker instance, producing a different set of bags, to compute other forms of decompositions. One possible extension would be to explore a design that instead produces all sets with a certain fractional cover number, thus allowing for fractional hypertree decompositions [53] to be computed.

In addition to this, we also provided some first optimisations for the basis check, which forms an important part of our design and an efficient implementation of the basis check is crucial for any effective implementation of our distributed algorithm. In order to see if our design is suitable to tackling difficult real-world instances, we also presented a prototype implementation written in the Go programming language, and we showed that it is already capable of solving many difficult instances that existing approach struggle with, even when running on the exact same machine. This shows the promise of the distributed algorithm for solving some of the hardest remaining instances of HyperBench, which have so far eluded any state of the art method.

Chapter 6

Conclusion

This last chapter will provide an overall summary of the main contributions of this thesis and explain how these contributions are advancing the field. We end this chapter on a discussion about future work and further lines of research in this area.

In Chapter 3 we began by first considering general improvements to algorithms computing GHDs. We provided a number of simplification rules which can reduce the search space and do not affect the correctness of the CheckGHD problem, thus improving the running times of all decomposition methods. As a next step, we designed a novel parallel algorithm for computing GHDs, based on a sequential algorithm from Fischl et al. [25]. We ended this chapter by demonstrating the practical applicability of our new algorithm by presenting an implementation of it, called BalancedGo and which is publicly available. We conducted a detailed experimental evaluation in which we compared our implementation against the state of the art in computing GHDs, and showed marked improvements in the number of instances which could be optimally solved in feasible time on the standard benchmark dataset HyperBench.

In Chapter 4, our investigation focused on the question of whether the Balanced Separator Approach we utilised in the previous chapter could be used for computing a different type of decomposition, namely HDs. As we have highlighted in Section 1.4, there are numerous challenges which make it impossible to use the exact same ideas as in our previous GHD algorithm. Instead, we introduce a novel normal form for HDs in Section 4.1 and based on this, we designed a method of computing an HD in such a way that we can split the problem of computing the HD into smaller parts *while retaining the information on where these parts will occur in the final HD we want to construct.* This required guessing the edge covers of two nodes at once, allowing us to exactly compute the bag of one of them. Through this idea, we managed to translate the Balanced Separator Approach to the setting of HDs and we presented our novel design for a parallel algorithm for computing HDs. As in the previous chapter, we provide a publicly available implementation of our algorithm, called log-k-decomp and we again showed in an experimental evaluation its strength relative to the state of the art in computing HDs.

We believe the advancements in these two chapters have already helped to make a strong argument for more in-depth exploration of the use of hypergraph decompositions in practical systems, such as query engines and constraint solvers. In fact, there is already initial research that shows the use of decompositions for these systems. Combined with the ability to quickly determine if a given CQ or CSP has low width and is thus of particular interest for exploiting its limited acyclicity, we hope that our work can help to provide significant improvements in solving CSPs and evaluating CQs in the real-world.

The last line of research investigated in this thesis is detailed in Chapter 5. We investigate the use of large scale distributed systems for the computation of hypergraph decompositions. We motivate this research by the presence of very challenging instances in the HyperBench dataset, which no existing decomposition method in the literature could solve optimally, including the two new methods we proposed in the earlier chapters. The shift to a new computation model required us to consider an entirely different approach, as the existing algorithms were designed with single shared-memory machines in mind. We found an interesting solution in the Candidate Tree Decomposition framework, presented by Gottlob et al. [39]. We adapted the algorithm presented there to our setting and presented in this chapter our design for a distributed algorithm for computing GHDs. We conclude the chapter by presenting a prototype implementation of this distributed algorithm, running on the Google Cloud Platform and we show some first promising results, by solving a number of challenging instances, which our two prior methods failed to solve, even when running on the same hardware.

This final line of research highlights the potential for exploring other computing platforms for finding decompositions, and in doing so it also opens up the question whether other problems, such also evaluating CQs and solving CSPs might be done effectively on the cloud as well.

6.1 Outlook

For future work, we envisage several lines of research: first, we want to further speed up the search for a first balanced separator. There is promising work on this from Schild and Sommer [87]. In addition to this we also want to speed up the search for a next balanced separator in case the first one does not lead to a successful decomposition. What makes these two goals challenging, is the fact that finding balanced separators has been shown to be a W[1]-hard problem by Marx [71] when parametrised by the size of the balanced separator. 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.

Our experiments in Section 4.7 suggest that there is significant potential in the study of metrics for hybrid approaches. In particular, how can we decide effectively when to switch from the balanced separation of `log-`k`-decomp` to the greedy heuristic guided method underlying `det-`k`-decomp`. This motivates a more in-depth study of hybridisation metrics in the future. In this work, we focused on using the size of subhypergraphs as the determining factor for when to switch. This is a rather crude measure, as even very large subhypergraphs might still be effectively decomposed by `det-k-decomp`. One possibility here would be to consider the treewidth of subhypergraphs as an initial heuristic for making this choice.

As part of this thesis, we have presented algorithms which use balanced separators for computing HDs and GHDs. As a next step, it would be interesting to explore whether balanced separators might also be used to effectively compute *fractional hypertree decompositions* (FHD) in parallel. There has been recent work which identified tractable cases for the computation of FHDs [38,39], which might be a useful starting point for such a line of research.

With HD computation for large and complex hypergraphs becoming practically feasible, one of the key challenges that block the use of HDs is quickly becoming less problematic. We therefore consider full integration of hypertree decompositions into existing database systems and constraint solvers to be a natural next step in this line of research. This would require implementing the Yannakakis algorithm on modern query engines. Initial work on this in the setting of incremental view updates for acyclic CQs provides evidence that the Yannakakis algorithm does help to reduce running times [58]. There has been initial work on this topic by Ghionna, Granata, Greco and Scarcello [29, 30], integrating the use of HDs into RDBMSs. Another research question is whether hypergraph decomposition (such as HDs or GHDs) can help to speed up existing CSP solvers. Initial work on this question by Amroun, Habbas and Singer and Amroun, Habbas and Aggoune-Mtalaa [7, 56] shows that a naive implementation of the algorithm by Gottlob, Scarcello and Leone [40] leads to inefficient methods, but the auhors also show the practical potential of more complex techniques which are still based on GHDs.

A related line of reseach in the use of hypergraph decompositions for practical applications is the question of finding *optimal weighted decompositions*. To give a brief explanation of this setting, we are given a hypergraph and some weight or *cost* function, which assigns to a given HD some numerical value, and *optimality* is understood as finding an HD of a given width which also has minimal cost, i.e. there does not exist any other HD of strictly lower cost. To motivate this setting, one could consider the cost function to reflect the estimated running time when using a given decomposition to evaluate a CQ. Initial work on this topic was done by Scarcello, Greco and Leone [84]. Given the new algorithms we have designed and presented in this thesis, the next question would be whether they could be extended to the setting of efficiently computing optimal weighted decompositions. This would also be of interest for the above question of using hypergraph decompositions in practice as well.

List of Figures

List of Tables

List of Algorithms

Bibliography

[1] Christopher R. Aberger, Susan Tu, Kunle Olukotun, and Christopher Ré. EmptyHeaded: A Relational Engine for Graph Processing. In *Proceedings of the 2016 ACM SIGMOD International Conference on Management of Data, SIGMOD Conference 2016*, pages 431–446, 2016.

[2] Christopher R. Aberger, Susan Tu, Kunle Olukotun, and Christopher Ré. Old Techniques for New Join Algorithms: A Case Study in RDF Processing. In *32nd IEEE International Conference on Data Engineering Workshops, ICDE Workshops 2016*, pages 97–102. IEEE Computer Society, 2016.

[3] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

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

[5] Foto N. Afrati, Manas R. Joglekar, Christopher Ré, Semih Salihoglu, and Jeffrey D. Ullman. GYM: A multiround Distributed Join Algorithm. In *Proceedings of the 20th International Conference on Database Theory, ICDT 2017*, volume 68 of *LIPIcs*, pages 4:1–4:18. Schloss Dagstuhl, 2017.

[6] Dmitri Akatov. *Exploiting Parallelism in Decomposition Methods for Constraint Satisfaction*. PhD thesis, University of Oxford, UK, 2010.

[7] Kamal Amroun, Zineb Habbas, and Wassila Aggoune-Mtalaa. A compressed Generalized Hypertree Decomposition-based solving technique for non-binary Constraint Satisfaction Problems. *AI Comm.*, 29(2):371–392, 2016.

[8] Franz Baader and Tobias Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998.

[9] Catriel Beeri, Ronald Fagin, David Maier, and Mihalis Yannakakis. On the Desirability of Acyclic Database Schemes. *J. ACM*, 30(3):479–513, 1983.

[10] Sally C. Brailsford, Chris N. Potts, and Barbara M. Smith. Constraint satisfaction problems: Algorithms and applications. *Eur. J. Oper. Res.*, 119(3):557–581, 1999.

[11] Ashok K. Chandra and Philip M. Merlin. Optimal Implementation of Conjunctive Queries in Relational Data Bases. In *Proceedings of the 9th Annual ACM Symposium on Theory of Computing*, pages 77–90. ACM, 1977.

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

[13] Rohit Chandra, Leonardo Dagum, Dave Kohr, Dror Maydan, Jeff McDonald, and Ramesh Menon. *Parallel Programming in OpenMP*. Morgan Kaufmann, 2001.

[14] Chandra Chekuri and Anand Rajaraman. Conjunctive query containment revisited. *Theor. Comput. Sci.*, 239(2):211–229, 2000.

[15] Katherine Cox-Buday. *Concurrency in Go: Tools and Techniques for Developers*. O'Reilly Media, Inc., 2017.

[16] Daniela S. Cruzes and Lotfi ben Othmane. Threats to Validity in Empirical Software Security Research. In *Empirical Research for Software Security*, pages 275–300. CRC Press, 2017.

[17] Rina Dechter. *Constraint Processing*. Elsevier, 2003.

[18] Alan A. A. Donovan and Brian W. Kernighan. *The Go programming language*. Addison-Wesley Professional, 2015.

[19] Tore Dybå and Torgeir Dingsøyr. Strength of Evidence in Systematic Reviews in Software Engineering. In H. Dieter Rombach, Sebastian G. Elbaum, and Jürgen Münch, editors, *Proceedings of the Second International Symposium on Empirical Software Engineering and Measurement, ESEM 2008, October 9-10, 2008, Kaiserslautern, Germany*, pages 178–187. ACM, 2008.

[20] M. Ayaz Dzulfikar, Johannes Klaus Fichte, and Markus Hecher. 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*, pages 25:1–25:23, 2019.

[21] Ronald Fagin. Degrees of Acyclicity for Hypergraphs and Relational Database Schemes. *J. ACM*, 30(3):514–550, 1983.

[22] Robert Feldt and Ana Magazinius. Validity Threats in Empirical Software Engineering Research - An Initial Survey. In *Proceedings of the 22nd International Conference on Software Engineering & Knowledge Engineering (SEKE'2010), Redwood City, San Francisco Bay, CA, USA, July 1 - July 3, 2010*, pages 374–379. Knowledge Systems Institute Graduate School, 2010.

[23] Johannes Klaus Fichte, Markus Hecher, Neha Lodha, and Stefan Szeider. An SMT Approach to Fractional Hypertree Width. In *Principles and Practice of Constraint Programming - 24th International Conference, CP 2018, Proceedings*, pages 109–127, 2018.

[24] Wolfgang Fischl, Georg Gottlob, Davide Mario Longo, and Reinhard Pichler. HyperBench: A Benchmark and Tool for Hypergraphs and Empirical Findings. In *Proceedings of the 38th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS 2019*, pages 464–480. ACM, 2019.

[25] Wolfgang Fischl, Georg Gottlob, Davide Mario Longo, and Reinhard Pichler. HyperBench: A Benchmark and Tool for Hypergraphs and Empirical Findings. *ACM J. Exp. Algorithmics*, 26:1.6:1–1.6:40, 2021.

[26] Wolfgang Fischl, Georg Gottlob, and Reinhard Pichler. General and Fractional Hypertree Decompositions: Hard and Easy Cases. In *Proceedings of the 37th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS 2018*, pages 17–32, 2018.

[27] Robert W. Floyd. Algorithm 97: Shortest path. *Commun. ACM*, 5(6):345, 1962.

[28] Eugene C. Freuder and Alan K. Mackworth. Constraint Satisfaction: An Emerging Paradigm. In *Handbook of Constraint Programming*, volume 2 of *Foundations of Artificial Intelligence*, pages 13–27. Elsevier, 2006.

[29] Lucantonio Ghionna, Luigi Granata, Gianluigi Greco, and Francesco Scarcello. Hypertree Decompositions for Query Optimization. In *Proceedings of the 23rd International Conference on Data Engineering, ICDE 2007*, pages 36–45. IEEE Computer Society, 2007.

[30] Lucantonio Ghionna, Gianluigi Greco, and Francesco Scarcello. H-DB: A hybrid quantitative-structural SQL optimizer. In *Proceedings of the 20th ACM Conference on Information and Knowledge Management, CIKM 2011*, pages 2573–2576. ACM, 2011.

[31] Georg Gottlob and Gianluigi Greco. Decomposing Combinatorial Auctions and Set Packing Problems. *J. ACM*, 60(4):24:1–24:39, 2013.

[32] Georg Gottlob, Gianluigi Greco, Nicola Leone, and Francesco Scarcello. Hypertree Decompositions: Questions and Answers. In *Proceedings of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS 2016*, pages 57–74. ACM, 2016.

[33] Georg Gottlob, Gianluigi Greco, and Francesco Scarcello. Pure Nash Equilibria: Hard and Easy Games. *J. Artif. Intell. Res.*, 24:357–406, 2005.

[34] Georg Gottlob, Martin Hutle, and Franz Wotawa. Combining hypertree, bicomp, and hinge decomposition. In *Proceedings of the 15th European Conference on Artificial Intelligence, ECAI 2002*, pages 161–165. IOS Press, 2002.

[35] Georg Gottlob, Matthias Lanzinger, Davide Mario Longo, Cem Okulmus, and Reinhard Pichler. The HyperTrac Project: Recent Progress and Future Research Directions on Hypergraph Decompositions. In *Integration of Constraint Programming, Artificial Intelligence, and Operations Research - 17th International Conference, CPAIOR 2020, Proceedings*, volume 12296 of *Lecture Notes in Computer Science*, pages 3–21. Springer, 2020.

[36] Georg Gottlob, Matthias Lanzinger, Cem Okulmus, and Reinhard Pichler. Experimental Data for log-k-decomp. Zenodo, November 2021. doi.org/10.5281/zenodo.6389816.

[37] Georg Gottlob, Matthias Lanzinger, Cem Okulmus, and Reinhard Pichler. Fast Parallel Hypertree Decompositions in Logarithmic Recursion Depth. In *Proceedings of the 41st ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS 2022*, pages 325–336. ACM, 2022.

[38] Georg Gottlob, Matthias Lanzinger, Reinhard Pichler, and Igor Razgon. Fractional Covers of Hypergraphs with Bounded Multi-Intersection. In *Proceedings of the 45th International Symposium on Mathematical Foundations of Computer Science, MFCS 2020*, volume 170 of *LIPIcs*, pages 41:1–41:14. Schloss Dagstuhl, 2020.

[39] Georg Gottlob, Matthias Lanzinger, Reinhard Pichler, and Igor Razgon. Complexity Analysis of Generalized and Fractional Hypertree Decompositions. *J. ACM*, 68(5):38:1–38:50, 2021.

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

[41] Georg Gottlob, Nicola Leone, and Francesco Scarcello. The Complexity of Acyclic Conjunctive Queries. *J. ACM*, 48(3):431–498, 2001.

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

[43] Georg Gottlob, Nicola Leone, and Francesco Scarcello. Hypertree Decompositions and Tractable Queries. *J. Comput. Syst. Sci.*, 64(3):579–627, 2002.

[44] Georg Gottlob, Nicola Leone, and Francesco Scarcello. Robbers, Marshals, and Guards: Game Theoretic and Logical Characterizations of Hypertree Width. *J. Comput. Syst. Sci.*, 66(4):775–808, 2003.

[45] Georg Gottlob, Zoltán Miklós, and Thomas Schwentick. Generalized Hypertree Decompositions: NP-hardness and Tractable Variants. *J. ACM*, 56(6):30:1–30:32, 2009.

[46] Georg Gottlob, Cem Okulmus, and Reinhard Pichler. Parallel Computation of Generalized Hypertree Decompositions. In *Proceedings of the 13th Alberto Mendelzon International Workshop on Foundations of Data Management*, 2019.

[47] Georg Gottlob, Cem Okulmus, and Reinhard Pichler. Fast and Parallel Decomposition of Constraint Satisfaction Problems. In *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, IJCAI 2020*, pages 1155–1162. ijcai.org, 2020.

[48] Georg Gottlob, Cem Okulmus, and Reinhard Pichler. Raw Data on Extended Experiments for BalancedGo. Zenodo, January 2021. doi.org/10.5281/zenodo.4411863.

[49] Georg Gottlob, Cem Okulmus, and Reinhard Pichler. Fast and Parallel Decomposition of Constraint Satisfaction Problems. *Constraints An Int. J.*, pages 1–24, 2022.

[50] Georg Gottlob and Marko Samer. A Backtracking-Based Algorithm for Hypertree Decomposition. *ACM J. Exp. Algorithmics*, 13, 2008.

[51] M. H. Graham. On The Universal Relation. Technical report, University of Toronto, 1979.

[52] Raymond Greenlaw and James H. Hoover. Parallel Computation: Models and Complexity Issues. In *Algorithms and Theory of Computation Handbook: General Concepts and Techniques*, page 28. CRC Press, 2nd edition, 2010.

[53] Martin Grohe and Dániel Marx. Constraint Solving via Fractional Edge Covers. *ACM Trans. Algorithms*, 11(1):4:1–4:20, 2014.

[54] William D. Gropp, Ewing L. Lusk, and Anthony Skjellum. *Using MPI - Portable Parallel Programming with the Message-Passing Interface, 3rd Edition.* Scientific and engineering computation. MIT Press, 2014.

[55] Marc Gyssens, Peter Jeavons, and David A. Cohen. Decomposing Constraint Satisfaction Problems Using Database Techniques. *Artif. Intell.*, 66(1):57–89, 1994.

[56] Zineb Habbas, Kamal Amroun, and Daniel Singer. 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, 2015.

[57] C. A. R. Hoare. Communicating Sequential Processes. *Commun. ACM*, 21(8):666–677, 1978.

[58] Muhammad Idris, Martín Ugarte, and Stijn Vansummeren. The Dynamic Yannakakis Algorithm: Compact and Efficient Query Processing Under Updates. In *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017*, pages 1259–1274. ACM, 2017.

[59] Mahmoud Abo Khamis, Ryan R. Curtin, Benjamin Moseley, Hung Q. Ngo, XuanLong Nguyen, Dan Olteanu, and Maximilian Schleich. Functional Aggregate Queries with Additive Inequalities. *ACM Trans. Database Syst.*, 45(4):17:1–17:41, 2020.

[60] Mahmoud Abo Khamis, Hung Q. Ngo, Christopher Ré, and Atri Rudra. Joins via Geometric Resolutions: Worst Case and Beyond. *ACM Trans. Database Syst.*, 41(4):22:1–22:45, 2016.

[61] Mahmoud Abo Khamis, Hung Q. Ngo, and Atri Rudra. FAQ: Questions Asked Frequently. In *Proceedings of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS 2016*, pages 13–28. ACM, 2016.

[62] Mahmoud Abo Khamis, Hung Q. Ngo, and Dan Suciu. What Do Shannon-type Inequalities, Submodular Width, and Disjunctive Datalog Have to Do with One Another? In *Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS 2017*, pages 429–444. ACM, 2017.

[63] Phokion G. Kolaitis and Moshe Y. Vardi. Conjunctive-Query Containment and Constraint Satisfaction. *J. Comput. Syst. Sci.*, 61(2):302–332, 2000.

[64] Tuukka Korhonen, Jeremias Berg, and Matti Järvisalo. Enumerating Potential Maximal Cliques via SAT and ASP. In *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI 2019*, pages 1116–1122. ijcai.org, 2019.

[65] Saidapet P. T. Krishnan and Jose L. Ugia Gonzalez. Google Cloud Pub/Sub. In *Building Your Next Big Thing with Google Cloud Platform: A Guide for Developers and Enterprise Architects*, pages 277–292. Apress, 2015.

[66] Vipin Kumar. Algorithms for Constraint-Satisfaction Problems: A Survey. *AI Magazine*, 13(1):32–44, 1992.

[67] Mohammed Lalou, Zineb Habbas, and Kamal Amroun. 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*, volume 589 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2009.

[68] Davide Mario Longo. PACE 2019 Hypertree Width Heuristic. Zenodo, May 2019. doi.org/10.5281/zenodo.3236369.

[69] David Maier. *The Theory of Relational Databases*. Computer Science Press, 1983.

[70] Riccardo Mancini, Srinivas Karthik, Bikash Chandra, Vasilis Mageirakos, and Anastasia Ailamaki. Efficient Massively Parallel Join Optimization for Large Queries. In *Proceedings of the 2022 ACM SIGMOD International Conference on Management of Data, SIGMOD Conference 2022*, pages 122–135. ACM, 2022.

[71] Dániel Marx. Parameterized graph separation problems. *Theor. Comput. Sci.*, 351(3):394–406, 2006.

[72] Dániel Marx. Tractable Structures for Constraint Satisfaction with Truth Tables. *Theory Comput. Syst.*, 48(3):444–464, 2011.

[73] Dániel Marx. Tractable Hypergraph Properties for Constraint Satisfaction and Conjunctive queries. *J. ACM*, 60(6):42:1–42:51, 2013.

[74] Lukas Moll, Siamak Tazari, and Marc Thurley. Computing hypergraph width measures exactly. *Inf. Process. Lett.*, 112(6):238–242, 2012.

[75] Bruce Momjian. *PostgreSQL: Introduction and Concepts*. Addison-Wesley Longman Publishing Co., Inc., 2001.

[76] David E. Narváez. Constraint Satisfaction Techniques for Combinatorial Problems. In *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence, (AAAI-18), the 30th innovative Applications of Artificial Intelligence (IAAI-18), and the 8th AAAI Symposium on Educational Advances in Artificial Intelligence (EAAI-18)*, pages 8028–8029, 2018.

[77] Adam Perelman and Christopher Ré. DunceCap: Compiling Worst-Case Optimal Query Plans. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, SIGMOD Conference 2015*, pages 2075–2076. ACM, 2015.

[78] Jorge Pérez, Marcelo Arenas, and Claudio Gutiérrez. Semantics and complexity of SPARQL. *ACM Trans. Database Syst.*, 34(3):16:1–16:45, 2009.

[79] Reinhard Pichler and Sebastian Skritek. Tractable counting of the answers to conjunctive queries. *J. Comput. Syst. Sci.*, 79(6):984–1001, 2013.

[80] Neil Robertson and Paul D. Seymour. Graph minors. III. Planar tree-width. *J. Comb. Theory, Ser. B*, 36(1):49–64, 1984.

[81] Neil Robertson and Paul D. Seymour. Graph Minors. II. Algorithmic Aspects of Tree-Width. *J. Algorithms*, 7(3):309–322, 1986.

[82] Walter L. Ruzzo. Tree-Size Bounded Alternation. *J. Comput. Syst. Sci.*, 21(2):218–235, 1980.

[83] Walter L. Ruzzo. On Uniform Circuit Complexity. *J. Comput. Syst. Sci.*, 22(3):365–383, 1981.

[84] Francesco Scarcello, Gianluigi Greco, and Nicola Leone. Weighted hypertree decompositions and optimal query plans. *J. Comput. Syst. Sci.*, 73(3):475–506, 2007.

[85] André Schidler and Stefan Szeider. Computing Optimal Hypertree Decompositions. In *Proceedings of the Symposium on Algorithm Engineering and Experiments, ALENEX 2020*, pages 1–11. SIAM, 2020.

[86] André Schidler and Stefan Szeider. Computing Optimal Hypertree Decompositions with SAT. In *Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence, IJCAI 2021*, pages 1418–1424. ijcai.org, 2021.

[87] Aaron Schild and Christian Sommer. On Balanced Separators in Road Networks. In *Experimental Algorithms - 14th International Symposium, SEA 2015, Proceedings*, volume 9125 of *Lecture Notes in Computer Science*, pages 286–297. Springer, 2015.

[88] Douglas Thain, Todd Tannenbaum, and Miron Livny. Distributed computing in practice: the Condor experience. *Concurrency - Practice and Experience*, 17(2-4):323–356, 2005.

[89] Edward P. K. Tsang. *Foundations of Constraint Satisfaction*. Computation in cognitive science. Academic Press, 1993.

[90] Susan Tu and Christopher Ré. DunceCap: Query Plans Using Generalized Hypertree Decompositions. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, SIGMOD Conference 2015*, pages 2077–2078. ACM, 2015.

[91] H. Venkateswaran. Properties that Characterize LOGCFL. *J. Comput. Syst. Sci.*, 43(2):380–404, 1991.

[92] Stephen Warshall. A Theorem on Boolean Matrices. *J. ACM*, 9(1):11–12, 1962.

[93] Mihalis Yannakakis. Algorithms for Acyclic Database Schemes. In *Proceedings of the 7th International Conference on Very Large Databases, VLDB 1981*, pages 82–94. VLDB, 1981.

[94] Mihalis Yannakakis. Node-Deletion Problems on Bipartite Graphs. *SIAM J. Comput.*, 10(2):310–327, 1981.

[95] Clement T. Yu and Meral Z. Özsoyoğlu. 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*, pages 306–312, 1979.