TU WIEN Informatics

On the Potential of Structural Decomposition of Database and AI Problems

DISSERTATION

submitted in partial fulfillment of the requirements for the degree of

Doktor der Technischen Wissenschaften

by

Davide Mario Longo, MSc
Registration Number 01529789

to the Faculty of Informatics

at the TU Wien

Advisor: Prof.Dr. Georg Gottlob
Second advisor: Prof.Dr. Reinhard Pichler

The dissertation has been reviewed by:

Francesco Calimeri	Zoltan Miklos

Vienna, 14th June, 2023

[redacted]
Davide Mario Longo

Erklärung zur Verfassung der Arbeit

Davide Mario Longo, MSc

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, 14. Juni 2023

[redacted]

Davide Mario Longo

Acknowledgements

I am deeply grateful to my advisor Georg Gottlob for guiding me through the challenges of my Ph.D. and for teaching me the value of relentlessness in the pursuit of truth. I would also like to thank my co-advisor Reinhard Pichler, whose expert help and mentorship have helped me grow as a researcher.

I am indebted to the entire DBAI team and the LogiCS people for backing me up during my doctoral studies. I am genuinely grateful to my co-authors for contributing to this work and being good research partners. My sincere gratitude goes to my friends and colleagues Matthias Lanzinger, Cem Okulmus, and Augusto B. Corrêa, who shared this journey with me and made it a rewarding experience.

I thank the Austrian Science Fund (FWF) for financially supporting my research through projects P30930-N35 and W1255-N23.

Finally, I would like to thank my family and friends for their constant love and encouragement. I am immensely grateful to my parents, Claudia and Carmine, for always encouraging me to chase my aspirations. My heartfelt gratefulness belongs to Sandra, who has been my anchor during the highs and lows of my Ph.D.

In conclusion, I am grateful to everyone who has played a significant role in my academic journey and helped me reach this important milestone.

Kurzfassung

Datenbanktheorie und Künstliche Intelligenz gehören zu den weitreichendsten Forschungsgebieten der Informatik. Obwohl klassische Probleme in diesen Bereichen, wie die Beantwortung konjunktiver Anfragen (Conjunctive Queries oder CQs) und die Lösung von Constraint Satisfaction Problemen (CSPs), oft natürlich auftreten, sind sie schwer lösbar. Der Grund dafür liegt in der potentiell komplizierten Struktur von CQ- und CSP-Instanzen, die durch *Hypergraphen*, eine Verallgemeinerung von Graphen, in denen Kanten mehr als zwei Knoten verbinden können, erfasst werden kann. Im Gegensatz zu den in polynomieller Zeit lösbaren Klassen von Instanzen mit einem zugrundeliegenden azyklischen Hypergraphen, stellen zyklische Probleminstanzen in der Tat ernsthafte Einschränkungen für Systeme dar. Die Einführung struktureller Zerlegungsmethoden, wie *generalized hypertree decompositions (GHDs)*, ermöglichte die Definition größerer Klassen von effizient lösbaren CQ- und CSP-Instanzen. Diese Methoden klassifizieren den Grad der Zyklizität von Hypergraphen durch das Konzept der Hypergraph *Breite*. Je geringer die Breite des Hypergraphen ist, desto mehr ähnelt er einem azyklischen Hypergraphen. Diese Ähnlichkeit spiegelt sich auch in der Komplexität des Problems wider: Je niedriger die Breite, desto einfacher ist das zugehörige Problem zu lösen. Die Berechnung von GHDs mit geringer Breite ist jedoch eine schwierige Aufgabe. Während die Theorie dieser Dekompositionen gut verstanden ist, wurde die Anwendung von Dekompositionen noch nicht hinreichend in die Praxis umgesetzt. Dieses Bestreben formt den Kern dieser Dissertation.

Wir beginnen diese Arbeit mit einer Studie über die Berechnung von GHDs für polynomiell lösbare Fälle. Die Berechnung von GHDs ist wünschenswert, weil sie geringere Breiten als andere Zerlegungen bieten. Obwohl frühere Arbeiten gezeigt haben, dass die Berechnung von GHDs für Hypergraphen mit begrenzter Kantenüberschneidung effizient ist, wurden nur rudimentäre Algorithmen vorgeschlagen. So gibt es zwar empirische Tests, um zu prüfen, ob eine GHD der Breite $\leq k$ eines Hypergraphen existiert, aber keinen Algorithmus, der eine entsprechende GHD berechnet. Ein solcher Test verwendet balancierte Separatoren, ein bekanntes Konzept der Graphentheorie, mit vielversprechenden Ergebnissen. Leider fehlt bis heute eine theoretische Fundierung dieser Methode für die Berechnung von GHDs. Daher entwickeln wir diesen Test zu einem neuen Algorithmus namens `BalSep`, der bei einem Hypergraphen H eine GHD von H mit der Breite $\leq k$ ausgibt, wenn sie existiert. Wir beweisen, dass `BalSep` korrekt und vollständig ist.

Weiters, gehen wir der Frage nach, ob BalSep und andere GHD-Algorithmen Hypergraphen aus realen CQs und CSPs effizient zerlegen können. Einschlägige frühere Arbeiten führten zur Erstellung von *HyperBench*, einer Sammlung von Hypergraphen von CQ- und CSP-Instanzen, die aus verschiedenen Anwendungsbereichen stammen. Während die Autoren von HyperBench zu dem Schluss kommen, dass sich reale Instanzen gut für Dekompositionsalgorithmen eignen, stellen wir fest, dass die Zusammensetzung von HyperBench nicht vielfältig genug ist, um diese Behauptungen allgemein zu stützen. In der Tat besteht dieser Datensatz nur zu einem Drittel aus CQs. Außerdem handelt es sich bei den meisten um einfache SQL-Abfragen, während komplexere Abfragen und andere beliebten Sprachen wie SPARQL vernachlässigt wurden. Da die Umwandlung dieser komplexen Anfragen in Hypergraphen eine neue Herausforderung darstellt, entwickeln wir eine neuartige Methode zur Extraktion der "maximalen konjunktiven Komponenten" aus einer Anfrage, die nicht rein konjunktiv ist. Dadurch erweitern wir *HyperBench* um eine vielfältige Auswahl von SPARQL- und komplexen SQL-Abfragen. Mit diesem erweiterten Datensatz durchgeführte Experimente zeigen, dass die Zerlegungsalgorithmen und insbesondere BalSep tatsächlich in der Lage sind, auch komplexere CQs effizient zu zerlegen.

Wir fahren mit der Untersuchung des Aktualisierens von GHDs in Reaktion auf Änderungen in der Probleminstanz fort. Bei Problemen wie der inkrementellen Constraint Satisfaction wird die Probleminstanz wiederholt verändert, um die schlussendliche Lösung zu berechnen. In diesem Fall muss ein GHD-basierter Solver jedes Mal, wenn sich die Instanz ändert, eine neue GHD berechnen. Auch wenn die Berechnung einer GHD mit geringer Breite eine lohnende Aufgabe ist, so ist die sehr häufige Berechnung von GHDs in diesem Szenario doch eine Herausforderung. Wir reagieren auf diese Herausforderung, indem wir das Problem der Aktualisierung einer GHD als Reaktion auf allgemeine Hypergraph-Modifikationen formalisieren. Leider erweist sich das Problem im Allgemeinen als genauso schwierig wie die Berechnung einer neuen Zerlegung. Dennoch entwickeln wir einen neuen Algorithmus, der die Berechnung einer GHD für die aktualisierte Aufgabe unter Verwendung der Kenntnis der alten Zerlegung erheblich beschleunigt. Ein Vergleich mit einem naiven Algorithmus, der eine GHD von Grund auf neu berechnet, zeigt, dass unsere Methode nicht nur extrem hohe Beschleunigungsfaktoren aufweist, sondern in den meisten Fällen auch die Breite nicht erhöht.

Schließlich erforschen wir die Verwendung von GHDs zur Grundierung klassischer Planungsaufgaben. Diese Aufgaben werden üblicherweise in einem Formalismus basierend auf Prädikatenlogik erster Ordnung formuliert. Da moderne Planer propositionale Aufgaben effizient lösen, grundieren sie die prädikatenlogische Darstellung um sie in einer propositionelle Form zu bringen, bevor sie nach einer Lösung suchen. Wenn die Aufgaben immer größer werden, wird hierbei jedoch die Grundierungsphase zu einem Engpass. Sogenannte Planning Grounder, spezielle Systeme welche diese Grundierungsphase umsetzen, arbeiten auf Basis von abgeschwächten Versionen der Aufgabe, in denen keine Negation auftritt. Auf Basis dieser abgeschwächten Version kann ein effizienter Plan zur Grundierung mittels Logikprogramme erzeugt werden. Bei großen Aufgaben haben diese Systeme allerdings

Schwierigkeiten, gute Auswertungspläne zu finden, um die Regeln des Logikprogramms auszuwerten. Diese Schwierigkeit ergibt sich aus der Größe und der Struktur der Aufgabe. Für Situationen wo diese Parameter für aktuelle Systeme zu hoch werden, entwickeln wir einen neuartigen Algorithmus, der, geleitet von einer GHD, die ursprünglichen Regeln in kleinere Regeln zerlegt, die leichter zu grundieren sind. Unsere empirische Evaluierung zeigt, dass dieser Ansatz die notwendige Zeit zur Grundierung erheblich reduziert.

Abstract

Database Theory and Artificial Intelligence are among the most far-reaching research areas in Computer Science. Although classical problems in these fields, like *answering conjunctive queries (CQs)* and *solving constraint satisfaction problems (CSPs)*, arise naturally in a simple formulation, they are intractable. The reason lies in the intricacy of the structure of CQ and CSP instances, which is neatly representable by *hypergraphs*, i.e., a generalization of graphs in which edges can connect more than two vertices. Indeed, in contrast to polynomial-time solvable classes of instances with an underlying acyclic hypergraph, cyclic problem instances pose severe limitations to systems. The introduction of structural decomposition methods, such as *generalized hypertree decompositions (GHDs)*, allowed the definition of larger classes of tractable CQ and CSP instances. These methods classify the degree of cyclicity of hypergraphs through the concept of *width*. The lower the hypergraph width, the more it resembles an acyclic hypergraph. The related instance is then easier to solve. However, computing low-width GHDs is a difficult task. While the theory of decompositions is well understood, the theoretical advantages of using decompositions have not yet been transferred into practice. This quest constitutes the central theme of this thesis.

We open this work with a study on the computation of GHDs for tractable cases. GHDs are desirable because they provide lower widths than other decompositions. Even though previous work demonstrated that computing GHDs for hypergraphs having bounded intersection size is tractable, only a few interesting yet rudimentary algorithms have been proposed. For instance, while empirical tests to check if a GHD of width $\leq k$ of a hypergraph exists, no algorithm computing a GHD exists. One such test uses balanced separators, a well-known concept in graph theory, with promising results. Unfortunately, a theoretical underpinning of this method is missing to date. Therefore, we develop this test into a new algorithm, called `BalSep`, that, given a hypergraph H, outputs a GHD of H of width $\leq k$ if it exists. We thus prove that `BalSep` is sound and complete.

We progress by asking if `BalSep` and other GHD algorithms can efficiently decompose hypergraphs from real-world CQs and CSPs. Relevant previous work led to the creation of *HyperBench*, a collection of hypergraphs of CQ and CSP instances originating from several applications' domains. While the authors of HyperBench conclude that real-world instances are well suited for decomposition algorithms, we recognize that the composition of the HyperBench is not varied enough to sustain these claims in general terms. Indeed,

this dataset is composed of only one-third of CQs. Moreover, most are simple SQL queries, to the detriment of more complex queries and other popular languages such as SPARQL. Since the transformation of these complex queries into hypergraphs presents us with a new challenge, we develop a novel methodology for extracting the "maximal conjunctive components" from a query that is not purely conjunctive. We thus extend *HyperBench* by including a more diversified sample of SPARQL and complex SQL queries. Experiments carried out on this extended dataset show that decomposition algorithms, and especially `BalSep`, manage to decompose even more complex CQs.

We move forward into studying the update of GHDs in response to instance modifications. In problems such as incremental constraint satisfaction, the solver repeatedly modifies the instance while computing a solution. In this case, a GHD-based solver must compute a new GHD every time the instance changes. Even though computing a low-width GHD is a worthwhile task, it is challenging. We respond to this challenge by formalizing the problem of updating a GHD in response to common hypergraph modifications. Unfortunately, we prove the problem to be as hard as computing a new decomposition anew. Yet, we develop a new algorithm that significantly speeds up the computation of a GHD of the updated task using knowledge of the old decomposition. A comparison against the naive algorithm computing a GHD from scratch reveals that not only our method has extremely high speed-up factors, but in most cases, it does not increase the width.

Finally, we research using GHDs to ground classical planning tasks. These tasks are typically formulated in a first-order formalism. Since state-of-the-art planners efficiently solve propositional tasks, they ground the first-order representation before searching for a solution. Nevertheless, the grounding phase becomes a bottleneck as the tasks get larger and larger. Planning grounders work on a logic program representing a relaxed version of the task where no negation occurs. When tasks are large, they struggle to find good join plans to evaluate the rules of the logic program. This difficulty is due to the number of joins and the structure of these joins. Consequently, we develop a novel algorithm that, guided by a GHD, splits the original rule into smaller rules that are easier to ground. Empirical evaluation shows that rules decomposition significantly reduces the grounding time if we perform the split judiciously.

Contents

CHAPTER 1

Introduction

This chapter introduces the context necessary to understand our research on structural decomposition methods. While we give a broad overview of the problems that most benefit from these methods, we present the main limitations of the current studies. They will lead us to our problem statement and allow us to formulate our research goal. As this chapter is intended for a broader audience, the first part follows a not-too-rigorous thread of narration. Later on, we will formally specify our research questions and enlist our contributions.

We accomplished this work in collaboration with many valued colleagues, which we will explicitly acknowledge in Section 1.4.

1.1 Difficult Problems in Databases and AI

Database Theory and Artificial Intelligence are two of the oldest and most fundamental research areas in Computer Science. The problems related to these disciplines often arise naturally and are simple to formulate, yet they are oftentimes difficult to solve. On this account, *answering of conjunctive queries* and *solving constraint satisfaction problems* constitute two relevant examples. Even though they are seemingly unrelated and indeed studied by different communities, they share a common reason for their complexity, which is related to the intricacy of the structure underlying instances of these problems. While we will delve into technicalities later in the thesis, we clarify here that our focus is on the structure underlying these problems and, thus, the connections between AI and Database Theory. Nonetheless, we devote our attention to the problems themselves. Indeed, in the last part of this thesis, we will show how we can proficiently use the techniques developed at the intersection of these two disciplines. Therefore, our study ultimately demonstrates that efficient algorithms exploiting the structural properties of instances solve hard problems. Nevertheless, instead of immediately jumping to conclusions, we first introduce our reader to the specific problems motivating this work.

Database systems are a core component of nearly all computer applications. Thus, it does not strike as surprising that extensive research has been done on them. While these systems present several challenges, such as transaction processing [78, 10, 109], distributing large-scale dataflows [41, 142], and producing interactive analytics from big data [83, 9], we focus here on *query answering*, i.e., the task of retrieving data from a database upon a user request. Following the standard database textbooks [5, 134], we treat query answering as a decision problem: given a database and a query, we want to know whether the result of the query is empty or not on the input database. In this setting, a query is a logical formula. The simplest yet most studied kinds of queries are *conjunctive queries (CQs)*, i.e., formulae consisting only of a conjunction of predicates, where neither disjunction nor negation appears. From a different perspective, CQs correspond to SQL *select-from-where* queries in which the WHERE clause contains only equalities between table attributes. These queries are often the target of query optimization algorithms as they frequently appear at the core of several applications such as query containment and incremental view maintenance [37, 34, 81]. Despite its formal simplicity, answering CQs is a hard problem [35].

Artificial Intelligence is another prominent field of research in Computer Science. Its roots trace back to the beginning of Computer Science with the long-term ambition of developing a thinking machine. Even though AI researchers have not yet achieved this goal, the discipline has developed into a research area where it is customary to find smart solutions for difficult problems. *Constraint Satisfaction Problems (CSPs)* are a powerful formalism that can capture many of the typical problems in AI. Indeed, CSPs have applications to several well-studied problems such as planning and machine learning [29, 138, 120]. Moreover, as CSPs offer a practical formalism to express combinatorial problems, they are widely used in Operations Research to solve problems such as scheduling and vehicle routing [98, 128]. Graph combinatorial problems are also typically formulated as CSPs [30, 42]. In a CSP, we have a set of variables and a set of constraints. While each variable has an associated domain, the constraints restrict the allowed combinations of values that the variables can be assigned. A solution for a CSP is an assignment of values to variables that does not violate any constraint. The price to pay for the high expressive power is the hardness of the problem [42].

We typically distinguish between easy and hard problems. The former are *tractable* because a computer can solve them in time polynomial in the input size. On the other hand, no polynomial algorithm is known for the latter, so they are *intractable*. Easy problems are grouped into the set $\mathcal{P}$, while hard ones are categorized in the set $\mathcal{NP}$. This distinction relies on the widely-believed assumption that $\mathcal{P} \neq \mathcal{NP}$ and thus no polynomial algorithm exists for hard problems, even though neither this claim nor its negation has been yet proven. In their general form, answering CQs and solving CSPs are $\mathcal{NP}$-complete problems. Intuitively, they belong to the class $\mathcal{NP}$ and are among the most difficult problems in this class.

Despite the differences in their formulations, these problems share more than it is possible to appreciate at first glance. In [97], Kolaitis and Vardi proved that answering CQs and solving CSPs are essentially the same problem; in particular, they can both be reformulated as the algebraic problem of finding a homomorphism between two relational structures. Moreover, we can translate these problems into logical formalism. In this setting, we perform the model checking of particular first-order formulae where only the existential quantifier and conjunction are allowed connectives while using the universal quantifier, disjunction, or negation is forbidden.

We can now start treating CQs and CSPs more uniformly, especially when discussing their complexity. It is well-known that not all CQs and CSPs are difficult. The tractability of acyclic CQs has been established in [36] in continuation of the work done by Yannakakis [140] and Qian [119] on query evaluation and containment for acyclic queries. On the other hand, tractable fragments of CSPs have been obtained by imposing restrictions on the constraints [103, 113]. Moreover, the complexity of evaluating acyclic boolean conjunctive queries has been successively refined in [70], where Gottlob et al. show that this problem is LOGCFL-complete, which means that this problem is highly parallelizable. This result holds for the satisfiability of CSPs as well. The structure of instances plays a crucial role in the tractability of these problems.

1.2 Hypergraphs and Structural Decomposition Methods

To better investigate acyclicity, we represent the structure of instances with *hypergraphs*, which are a generalization of graphs where edges are not restricted to be binary. More precisely, a hypergraph H is a pair $(V(H), E(H))$ where $V(H)$ is a set of vertices and $E(H) \subseteq 2^{V(H)}$ is a set of hyperedges over $V(H)$. In the case of a conjunctive query Q, the hyperedges correspond to the relation predicates in Q, while the vertices are defined as the set of variables occurring in Q. A CSP P induces a hypergraph as follows: the variables of P become vertices, and each constraint C of P is a hyperedge connecting all vertices corresponding to variables in C.

In contrast to graphs, for which acyclicity is defined unequivocally, there are several reasonable ways to define acyclicity for hypergraphs [45]. In order of increasing generality, the most popular notions are γ-, β-, and α-acyclicity. In [31], Brault-Baron argued that acyclicity is characterized in two ways: (1) by forbidding certain substructures like cycles; and (2) through reducibility to the empty hypergraph by repeatedly applying certain reduction rules. The author also showed that these two characterizations coincide. Since the most fruitful notion of acyclicity in the database area has been proven to be α-acyclicity, we focus on this here. A characterization of α-acyclicity close to the sense of (1) has been given in [19], where the authors argued that acyclic database schemes have desirable properties. Alternative characterizations of type (2) have been extensively used to solve database problems proficiently. In [77, 141], Graham, Yu, and Özsoyoglu independently introduced the GYÖ algorithm, which tests in polynomial time whether a hypergraph is acyclic or not. Moreover, Yannakakis characterized CQ acyclicity based on *join trees* [140]. The same paper showed that it is possible to compute a join tree in polynomial time, but linear-time procedures exist as well [130]. Similarly, Dechter and

Pearl gave an analogous tree-like characterization for CSPs [43].

The success of acyclicity as a structural tractability criterion motivated further research on larger islands of tractability. In particular, researchers spent efforts on the tree-based characterizations of acyclicity, which produced a variety of measures of the degree of cyclicity of hypergraphs, mostly called widths. A proper generalization of graph acyclicity is *treewidth* [123, 122], which is associated to *tree decompositions*. These are a tree-like representation of a graph suggesting how to split the original problem into smaller subproblems that can be solved independently from each other and whose solutions are then combined to obtain a solution to the original problem. The lower the width of the decomposition, the easier it is to solve the problem at hand. Acyclic graphs have a treewidth equal to 1. Computing a tree decomposition of bounded width, say $\leq k$, with k constant, is feasible in linear time [24]. Even though the treewidth of a hypergraph can be computed by decomposing its primal graph, treewidth does not properly generalize hypergraph acyclicity. There are indeed classes of acyclic hypergraphs whose treewidth is unbounded. A proper generalization of hypergraph acyclicity was proposed in [37] under the name of *query width*. Nevertheless, computing a query decomposition of bounded width is hard for any $k > 3$ [72].

At that point, a new width measure was introduced by Gottlob, Leone, and Scarcello in [72], namely *generalized hypertree width (ghw)*. Analogously to query width, *ghw* generalizes hypergraph acyclicity, but it can be smaller than the query width of the same hypergraph. Nonetheless, generalized hypertree decompositions (GHDs) are still hard to compute for any $k \geq 2$ [73, 53]. This limitation was overcome in the same article by adding a structural constraint on the shape of GHDs. The resulting width is called *hypertree width (hw)*. It has been proved that computing a hypertree decomposition (HD) of width $\leq k$ is tractable [72]. Moreover, given a conjunctive query Q and an HD, it is possible to compute the answer of Q in polynomial time in combined complexity [137] by using any algorithm for join trees like the Yannakakis algorithm [140]. Therefore, hypertree width enjoys all the three desirable characteristics enlisted in [64]: queries of $hw \leq k$ include the acyclic ones; queries of $hw \leq k$ can be identified in polynomial time (for fixed k); and queries of $hw \leq k$ can be answered in polynomial time (for fixed k). The drawback of HDs compared to GHDs is that for each hypergraph H, $ghw(H) \leq hw(H)$ meaning that GHDs potentially lead to faster query answering.

Over time more widths have been proposed to generalize acyclicity even further. A notable example is *fractional hypertree width (fhw)* [79], for which it holds that $fhw(H) \leq ghw(H)$, for any hypergraph H. Nevertheless, computing a fractional hypertree decomposition of width $\leq k$ is intractable for any rational $k > 1$ [53, 69]. Later, a more general notion of width appeared: submodular width [106]. While queries enjoying bounded submodular width are fixed-parameter tractable when considering query size as a parameter, whether such queries are efficiently recognizable is unknown. All the widths introduced until now are purely structural, which means that they only take into account the hypergraph structure of the instance at hand. A hybrid width would also consider the database on which the query must be answered. One such parameter is *joinwidth* [55], which generalizes *fhw*, allows for tractable query answering, and fixed-parameter tractable recognizability. In the same spirit of involving the database in the computation of an appropriate hypertree decomposition, Ganian et al. [56] introduced the *threshold hypertree width*. We focus, in this thesis, on purely structural widths, primarily on *ghw* and *hw*.

We deal with *ghw* and *hw* because efficient algorithms exist to compute the respective decompositions. Since computing HDs of width $\leq k$ is tractable, many efficient implementations are available. The first top-down algorithm for this problem appeared in [76]. Contrary to the first bottom-up implementation in [71] computing optimal HDs, this algorithm scales better with the increasing size of the input hypergraph. The problem of computing an HD of optimal width is indeed more challenging. For this case, Schidler and Szeider proposed some algorithms based on SMT solving and SAT that work well in practice [125, 126]. In contrast to HDs, computing GHDs is hard in general. Nevertheless, we already discussed that a GHD could potentially lead to better decompositions, so research has been done to find tractable fragments of the problem of computing a GHD of width $\leq k$. In [53], Fischl et al. showed that by exploiting some specific properties of hypergraphs concerning the intersection size between edges, it is possible to compute GHDs efficiently for large classes of hypergraphs. This discovery led to the implementation of several algorithms which produce GHDs. Fischl et al. proposed a sequential implementation in [50]. Later, a parallel implementation has been made available [75], which proved extremely suitable even for instances having a large width. Also, the SMT solver approach proposed by Fichte et al. [47] can be adapted to compute optimal GHDs. Consequently, the interest in using GHDs and HDs in practice is getting higher.

1.3 Decompositions in Real Systems

Structural decomposition methods have been originally defined as a theoretical construct to define larger classes of tractable conjunctive queries. Yet, the concept has become so well-established that decompositions carved their way into applied research and commercial systems. Here, we give examples of systems successfully using decompositions to solve database and AI problems.

When using decompositions to solve real instances, there is more at hand than the bare structure of the instance. For example, when answering a query on a database system, the choice of a good query plan is based on quantitative information about the database, such as the number of tuples contained in the tables involved in the query and other relevant statistical data. In [124], Scarcello et al. introduced weighted hypertree decompositions as an extension of HDs weighted by quantitative factors. These decompositions have been used as a basis to produce optimal query plans in the system implemented by Ghionna et al. [61, 62], where they proved the effectiveness of this approach for a narrow selection of queries from the well-known TPC-H benchmark [132].

Decomposition-based techniques have also been implemented in commercial database systems. LogicBlox [12] is a general-purpose database system challenging the belief that specialized systems are necessarily better than general-purpose ones. Motivated by the fact that modern systems often need to integrate several specialized components, they decided to design a new architecture for a general-purpose database system that uses the latest research discoveries. Their implementation of novel worst-case optimal join algorithms based on fractional hypertree decompositions [16, 94, 95, 112, 93] proved to be competitive with more established commercial systems. The algorithms used in the system even exploit submodular width [96]. Decomposition-based techniques have also been employed in graph processing systems such as EmptyHeaded [3, 4]. In this system, the query compiler produces query plans based on fractional hypertree decompositions [115, 133]. An extension of EmptyHeaded oriented to business intelligence and linear algebra is LevelHeaded [2]. This system uses the same decomposition techniques as EmptyHeaded but performs better in more specialized domains. GHDs have been used also to answer queries in a distributed model a la MapReduce in [8].

To a lesser extent than database systems, decompositions have been used to solve constraint satisfaction as well. In [11, 82], a forward-checking algorithm based on hypertree decompositions was presented. With their implementation, the authors improve on well-known memory problems due to the materialization of big constraint relations. In a similar spirit, parallel algorithms based on HDs were introduced in [99]. Karakashian et al. [91] proposed a rewriting technique based on hypertree decompositions introducing redundant constraints to improve the performance of search algorithms.

1.4 Research Questions and Main Results

Based on solid theoretical foundations, the practice of computing and using structural decomposition methods flourishes. Consequently, new questions about the relationship between theory and the application of decompositions arise. In this section, we introduce the ones addressed by this thesis and present the results of our investigation. This consists of four thematic areas concerning the formalization of a new algorithm computing GHDs, the significant extension of a benchmark for testing decomposition algorithms, the update of GHDs upon changes to the original instance, and the use of GHDs to ground planning tasks.

The contributions made by this thesis are the result of joint work with various colleagues. The articles on which these results are based will be cited in the corresponding subsections, as will the colleagues who contributed to this work.

1.4.1 The Computation of GHDs through Balanced Separators

This work was carried out in collaboration with Wolfgang Fischl, Georg Gottlob, and Reinhard Pichler. The main contributions of this section are based on [52], published in the ACM Journal of Experimental Algorithmics (JEA) in 2021. This article is an extended version of [50], presented at the ACM Symposium on Principles of Database Systems (PODS) 2019.

In contrast to HDs, for which the Check(hw, k) problem is tractable for any fixed $k \geq 1$, computing a GHD of width $\leq k$ is a hard problem, for any $k \geq 2$ [73, 53]. While bounded *hw* implies bounded *ghw* and vice versa [7] and, thus, these two measures characterize the same class of tractable CQs and CSPs, being able to use a decomposition of lower width proved to be beneficial in practice [3]. It is indeed well known that for each hypergraph H, $ghw(H) \leq hw(H)$, thus making the availability of a low-width GHD desirable.

Fischl, Gottlob, and Pichler accepted the challenge of identifying tractable fragments for GHD computation and proved in [53] that the Check(ghw, k) problem is tractable for all classes of hypergraphs enjoying the *bounded multi-intersection property*. In particular, the problem is fixed-parameter tractable when parameterized by *intersection size*, defined as the maximum intersection size between any two edges of a hypergraph. This property has been used by Fischl et al. in [50] to implement several algorithms for computing GHDs of bounded width efficiently.

Notably, Fischl proposed in [49] an empirical procedure to check, given a hypergraph H and an integer $k \geq 1$, whether H allows for the computation of a GHD of H of width $\leq k$. This test based on so-called *balanced separators* has been proficiently used there for numerous experiments. Later on, in [50], a similar procedure for the more general case of computing a GHD was delineated. Unfortunately, a formal definition of this algorithm and proof of its correctness are missing.

Research Challenge: Formalize the GHD algorithm based on balanced separators and study its properties.

To this aim, we combine the idea of using intersection size for fixed-parameter tractable GHD computation in [50] with the balanced-separators-based method by Fischl [49]. The result is a revised version of the `BalSep` algorithm in [49, 50] that, given a hypergraph H and an integer $k \geq 1$, computes a GHD of H of width $\leq k$ in polynomial time.

Main Result 1: We formalize the `BalSep` procedure originally outlined in [49, 50] and extend it to an algorithm for actually computing GHDs.

A rigorous formal analysis is necessary because of how `BalSep` constructs GHDs. Indeed, this algorithm differs from the standard method of computing decompositions in that some "artificial" edges, which were not present in the original hypergraph, are introduced during the computation. Moreover, the subhypergraphs generated during the computation are decomposed in a "sparse" order and must be assembled correctly into a GHD of the original hypergraph. It is thus not clear whether `BalSep` is correct. These challenges require a formal theory supporting and justifying the choices made in designing `BalSep`.

Main Result 2: We extend the definition of GHD to the case of *extended (sub)hypergraphs* where certain "special edges" must adhere to specific rules.

Main Result 3: We prove that the `BalSep` algorithm is sound and complete.

1.4.2 Benchmarking Decomposition Algorithms

This research was conducted with Wolfgang Fischl, Georg Gottlob, and Reinhard Pichler. The results presented in this section come from the article [52] published in the ACM Journal of Experimental Algorithmics (JEA) in 2021. The experiments carried out here expand the ones published in [50] and presented at the ACM Symposium on Principles of Database Systems (PODS) 2019. They were also presented at the Alberto Mendelzon International Workshop on Foundations of Data Management in 2019 [51]. This section also contains novel methods for analyzing complex SQL queries, which appeared neither in [50] nor in [49] and thus constitute original content. Related content about the challenges of benchmarking decomposition algorithms also appeared in this survey [66], published at the Integration of Constraint Programming, Artificial Intelligence, and Operations Research Conference (CPAIOR) 2020.

Designing a benchmark for evaluating decomposition algorithms is fundamental in bridging the gap between the theory of structural decomposition methods and the practice of answering queries and solving CSPs. In [50], Fischl et al. collected numerous real-world CQs and CSPs to discover if their hypergraphs have favorable structural properties for the use of decomposition algorithms. At the same time, they wanted to evaluate the performance of the GHD algorithms they proposed in the same article. Their work led to the creation of *HyperBench*, a collection of thousands of hypergraphs and algorithm runs. The investigation on HyperBench provided conclusive evidence that real-world hypergraphs have favorable properties for tractability and that the algorithms used are well-suited for these instances.

Nevertheless, while HyperBench certainly contains good examples of real-world instances, its composition is problematic. Only about a third of the hypergraphs originate from CQs, most of which are SQL queries. Moreover, we recognized that most of these queries were written in a simplified format using only a restricted subset of the language. Similarly, other query languages are not well represented in the dataset. For instance, only about 7% of the total query hypergraphs in HyperBench come from SPARQL queries, a popular query language for graph databases.

Research Challenge: Extend the HyperBench by providing a representative view of the query answering landscape.

The main challenge with complex SQL queries is understanding which queries are rewritable into a simple conjunctive form. Looking at the SQL language, we identify as simple conjunctive queries those SELECT-PROJECT-JOIN queries where no nested subqueries are present. The class of complex SQL queries is thus composed of all those queries containing views, nested subqueries, and other advanced features. Since the presence of these constructs does not necessarily make a query semantically non-conjunctive, we can rewrite them into a simple form. Nevertheless, not all complex SQL queries are conjunctive. Thus, we must determine how to handle them.

> **Research Challenge:** Devise a methodology for meaningfully translating complex SQL queries into hypergraphs.

We propose to split complex SQL queries into a collection of one or more simple conjunctive queries representing the original query. We achieve this goal by identifying maximal conjunctive components, i.e., maximal conjunctive subqueries that can be evaluated independently. Intuitively, given a collection of such simple queries, it should be possible to obtain the result of the original query by combining the answers of the simpler conjunctive parts. Furthermore, we provide an implementation of this method.

> **Main Result 4:** We devise an algorithm to extract simple conjunctive components from complex SQL queries.

> **Main Result 5:** We implement a tool to transform complex SQL queries into hypergraphs.

Searching for complex SQL queries, we direct our attention towards the TPC-DS benchmark [131]. This comprehensive set of industry-relevant queries was designed to benchmark modern general-purpose decision support systems and is therefore composed of a collection of complex and carefully hand-crafted SQL queries. Our new algorithm extracts 228 hypergraphs from the original 113 complex SQL queries. Additionally, we work on the front of SPARQL queries by examining a vast array of queries posed to Wikidata and released by Malyshev et al. [104]. These queries are particularly interesting for this work, not only because they represent a different application domain than the queries already in HyperBench, but they are also a mix of well-structured synthetic queries and unpolished human queries. We examine the hypergraphs of these queries, previously studied by Bonifati et al. in [28]. We identify 354 distinct hypergraphs from a pool of 273947 unique queries. As a result of our efforts, we collect 578 new hypergraphs from SPARQL and complex SQL queries.

> **Main Result 6:** We provide an extended version of HyperBench that includes hypergraphs originating from 1613 queries and 2035 CSPs.

We want to know if GHD algorithms perform well on real-world hypergraphs, particularly our revised version of `BalSep`. However, given the different composition of HyperBench, we question whether the conclusions drawn in [50] still hold. Indeed, the low width and intersection sizes of CQs could follow from the simple structure of the queries in the original dataset.

> **Research Challenge:** Discover if the width and intersection sizes of the new hypergraphs are low enough to use GHD algorithms in practice.

We replicate the experiments conducted by Fischl in [49] on our extended version HyperBench. To begin, we examine the structural properties of the new hypergraphs, such as *hw* and (multi-)intersection sizes. Moreover, we aim to evaluate the performance of GHD algorithms and assess the behavior of `BalSep`.

Main Result 7: We establish that the new hypergraphs have low width and intersection sizes.

Main Result 8: We empirically prove that `BalSep` is well-suited for decomposing hypergraphs.

1.4.3 Updating GHDs after Instances Modifications

This section is based on the work carried out in [65] with Georg Gottlob, Matthias Lanzinger, and Cem Okulmus, published in the ACM Journal of Experimental Algorithmics (JEA) in 2023.

Solving CSPs as well as answering CQs with decompositions can be beneficial, especially if the width of the decompositions used is low. For example, Aberger et al. [3] use low-width GHDs to speed up query answering. It is thus worth investing resources in computing a low-width GHD, with the additional advantage that a GHD can even be reused several times to answer the same query over an updated database. On the other hand, computing low-width GHDs is a hard task. Hence, we want to avoid the computation of a new GHD whenever possible. Since a decomposition is tightly linked to the hypergraph of the instance, unfortunately, any slight modification of the instance at hand implies that the GHD computed using many precious resources must be recomputed afresh. This results in a loss on the investment previously made.

Research Challenge: Understand how hypergraph changes affect its decompositions.

There are several problems where instances are frequently modified. For instance, to solve *incremental constraint satisfaction*, the constraint solver needs to handle mutable sets of variables [127] or constraints [54]. Similarly, a compositional modeling problem consists in synthesizing the most appropriate model of a physical system for a given analytical query [46]. The construction of the "best" model passes through several phases in which the model is iteratively refined by modifying constraints. Here, support during the modeling process is needed to better understand the impact of the modifications on the complexity of solving the problem using the resulting model. Motivated by all these applications, we study the problem of updating GHDs in response to instance updates in the setting of constraint satisfaction.

Research Challenge: Examine the typical ways in which a CSP is modified.

Main Result 9: We define a framework for constraint modification based on elementary changes performable on CSPs. Moreover, we detail how these modifications affect the hypergraph of the CSP at hand.

Changes to a CSP must be reflected on the GHD of the original instance. At first glance, it is not obvious whether a simple update to the decomposition structure is enough or whether more radical modifications are needed. At this point, it is even unclear if the knowledge of the old GHD helps compute a decomposition for the updated instance.

Research Challenge: Formalize the problem of recomputing a GHD of a modified instance and identify its complexity.

Main Result 10: We introduce the novel SEARCHUPDATEGHD problem.

Main Result 11: We pinpoint the complexity of SEARCHUPDATEGHD for a set of relevant elementary modifications. The problem is not solvable in polynomial time under standard assumptions for most modification classes.

Even though updating GHDs is difficult, we investigate ways to solve the problem in a more practice-oriented way. When a hypergraph is modified, we look at the available GHD and test whether some nodes are still valid considering the new hypergraph. If a substantial portion of the tree is still legal after modifications, there is no reason to discard this decomposition and compute a completely new one. It might be wiser to recompute only the small parts that are not valid anymore. We, therefore, devise a theory for updating GHDs.

Research Challenge: To what extent recompute an old GHD when the related instance is modified?

Main Result 12: We introduce a general framework for updating GHDs upon any kind of modification. This allows us to identify the minimal amount of the original GHDs to be updated.

We now put into practice what we learned while formulating our theory. Therefore, we analyze the existing algorithms for the computation of GHDs to understand whether we can extend them to the case of GHD updates. Since most of these, such as [76, 52, 74], follow the same top-down schema, we refine them to address the case where a partial decomposition is given as input. We define a general schema addressing updates and show that integrating these new techniques can be easily made.

Research Challenge: Can we practically combine our theoretical framework with the state-of-the-art algorithms for computing GHDs?

Main Result 13: We devise and implement an algorithmic update schema for updating GHDs that can be instantiated with any top-down algorithm for computing GHDs.

Main Result 14: Our implementation outperforms classical methods with average speedups between factors 6 and 50.

1.4.4 Grounding Datalog programs in Classical Planning

The work presented in this section was conducted with Augusto B. Corrêa, Markus Hecher, Malte Helmert, Florian Pommerening, and Stefan Woltran. It resulted in the article [38], accepted at the International Conference on Automated Planning and Scheduling (ICAPS) 2023. In addition to the tree-decomposition-based method to ground classical planning tasks presented in the article, we propose an original algorithm using generalized hypertree decompositions.

Query answering appears even in challenging AI problems such as classical planning. Given an initial state of the world I and a set of action schemas $\mathcal{A}$, we must compute an action sequence $\pi = \langle A_1, \ldots, A_n \rangle$, where $A_i \in \mathcal{A}$, to transform I into a desired goal state G. The solution π is a *plan*. Given a planning task Π, deciding if Π admits a plan is PSPACE-complete and thus extremely difficult [32]. A planning task is typically described in a first-order formalism. If the planner searches for a plan on this representation, we talk about *lifted planning problems*. Corrêa et al. proved that the *lifted successor generation* problem is equivalent to query answering in a database setting [39]. Given a state of the world s and a set of action schemas $\mathcal{A}$, generating all successor states of s requires checking which action schemas from $\mathcal{A}$ are applicable in s. Each action schema A has a precondition $pre(A)$, i.e., a logic formula that, if satisfied in s, establishes that A is applicable in s.

Nonetheless, state-of-the-art planners ground the task into a propositional representation before looking for a plan [26, 88, 86, 92]. In this case, we talk about *grounded planning problems*. Searching for a plan on a grounded representation is exponentially easier than in the lifted case [44]. Yet, the grounding phase remains an obstacle, as the size of the grounding could be exponential in the size of the task. Perhaps the most used planning grounder is Helmert's algorithm [87] designed for his planner Fast Downward [86]. Helmert reports that grounding consumes 70% of the total preprocessing time on well-established benchmarks. As planning tasks get larger and larger, grounding becomes a performance bottleneck. Since grounding involves solving the lifted successor generator problem multiple times, we believe that database techniques can prove beneficial in this setting.

Research Challenge: Improve planning grounders using database-inspired technology.

Answer Set Programming (ASP) [17] is an example of the proficient use of grounders. Two examples are relevant to our work. Morak and Woltran proposed in [110] a rewriting method for logic programs based on tree decompositions obtaining promising results. Calimeri et al. [33] improved this method for the *DLV* system [102]. Unfortunately, examining this method outside its context is difficult since *DLV* tightly integrates the grounding and search phases. On the other hand, Morak and Woltran's approach is abstract enough to be tested in planning.

Nevertheless, when grounding a planning task, we generate a logic program corresponding to a relaxed version of the original task, which does not contain any negation and is thus simpler than a typical ASP program. Since the tree decomposition method was designed to ground full ASP programs and it is well-known that generalized hypertree decompositions have lower upper bounds for the size of intermediate results, we believe that a more tailored approach is desirable. Therefore, we transform each rule of the logic program into a hypergraph and compute a GHD. Then, guided by the decomposition, we split the original rule into smaller rules to reduce intermediate results.

Main Result 15: We devise a novel method based on GHDs to split the rules of a logic program and reduce the size of intermediate results.

At this point, we empirically compare different rewriting methods based on rules decomposition. Using the original logic program as a baseline, we compare our GHD method against Morak and Woltran's TD approach and against the cutting-edge decomposition method used by Helmert in Fast Downward. Since the *ghw* of the rules is always lower than their *tw*, we expect our method to perform best. Nonetheless, the experiments show that the TD method performs best, but only slightly better than grounding the original program. Surprised by these results, we investigate the reasons for this behavior.

Research Challenge: Determine the fundamental parameters for the success of task decomposition.

Looking at the formulation of the logic program of the relaxed task, we notice that the presence of the so-called action predicates artificially introduces a clique between all vertices of the rule's primal graph and an edge covering all vertices of the rule's hypergraph. While this makes the hypergraph trivially acyclic, it highly increases the treewidth. For this reason, the program decompositions are too similar to the original program and do not improve performance. On the other hand, action predicates are not necessary for the grounding phase and can be safely removed.

Main Result 16: We significantly increase the number of grounded tasks by removing action predicates from the grounding.

Main Result 17: We considerably decrease the grounding time by combining the tree decomposition split and the removal of action predicates.

While removing action predicates leads to more satisfactory results, we observe that, counterintuitively, the TD method works better on average than our GHD method despite the higher widths. However, there are two classes of instances where the performance of our GHD method significantly deviates from that of the other methods. In the first case, our method is two orders of magnitude faster than the others on average. Here, our decomposition forces the grounder to compute highly selective joins early, speeding up the whole grounding. In the second case, our method is two times worse than other methods on average. Unfortunately, our decomposition forces the grounder to materialize an expensive cartesian product otherwise avoided by other methods.

Main Result 18: We discover that join selectivity is a better predictor of grounding time than width, regardless of the type of decomposition.

1.5 Overview of the Study

This thesis consists of six additional chapters conceptually divided into three parts. In the first part, composed by Chapter 2, we present the essential concepts needed to develop this thesis. This chapter contains a gentle yet formal introduction to hypergraphs, various forms of decompositions, and a general top-down schema used by most previous algorithms for decomposing hypergraphs.

The second part consists of Chapter 3, Chapter 4, Chapter 5, and Chapter 6, which detail the work carried out to respond to the research challenges presented in the previous section. The formalization of an algorithm computing GHDs based on balanced separators is presented in Chapter 3. This algorithm is used in Chapter 4 to understand how difficult are instances typically found in practice. This chapter also presents novel methods to process complex queries which are not purely conjunctive yet can still be split into conjunctive parts. The `BalSep` algorithm and the benchmark constructed in these first two core chapters are used in Chapter 5 to study the problem of updating an already computed GHD when the corresponding decomposed instance is modified. Chapter 6 shows how GHDs can be used to improve the performances of grounders in the context of classical planning.

Chapter 7 constitutes the final part of this thesis. While recapitulating the results achieved in this study, we offer a reflective evaluation of the performed work and suggest in which direction further research could follow.

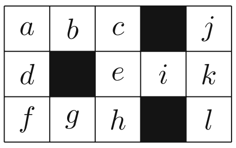
CHAPTER 2

Preliminaries and Definitions

In the introductory chapter we mentioned the research challenges we faced and presented an overview of the results we achieved in this thesis. While we introduced the reader to the research field and mentioned the problems motivating our study, we did not formally define all the concepts we use in this work. Yet, a definition of the main terms and a clarification of the differences between different constructs is necessary.

In this chapter, we provide the most important definitions necessary to proceed with the thesis. We formally define hypergraphs and show how they stem from real-world problems. We also present the concept of *structural decomposition methods* and introduce two particular methods on which we extensively focus in this thesis, namely *hypertree decompositions* and *generalized hypertree decompositions*. In doing so, we give additional background for those concepts that require particular attention. Furthermore, important results from the literature will also be presented where appropriate. All of these notions will constitute the vocabulary we will use for the rest of the thesis.

Structurally, this chapter is divided into four sections. First, we define the classical problems motivating this study, i.e, answering conjunctive queries and solving constraint satisfaction problems. We also show how hypergraphs originate from the instances of these problems. Next, we define the different kinds of structural decomposition methods and the relative widths. An important part of this thesis concerns the computation of decompositions, thus, we introduce a general top-down method of computing decompositions, which appears several times in practice. Finally, we recapitulate important structural properties of hypergraphs that make the computation of generalized hypertree decompositions polynomial.



(a) Puzzle P.

W		
d	o	g
o	d	d
g	o	d

(b) Relation W.

Figure 2.1: A crossword puzzle P and the relation W of possible words. We want to fill every contiguous horizontal or vertical line of white cells with words from W. If two lines intersect, the words assigned to these lines must intersect in the right positions.

2.1 CQs, CSPs, and Hypergraphs

Conjunctive queries (CQs) are the simplest, yet fundamental, kind of queries that can be posed to a database. A query in this class can be easily represented as first-order logic formula in which only the connectives $\{\exists, \wedge\}$ are allowed and $\{\forall, \vee, \neg\}$ are disallowed. A *relational signature* σ is a finite set of relation symbols, each of which is associated to an arity. A *database* D over a signature σ consists of a finite *domain* Dom and a relation R^D for each relation symbol R in σ. Thus, using the comma to denote the logical AND, a conjunctive query q is a conjunction of literals of the kind

$$q(y) \leftarrow R_1(x_1), \ldots, R_m(x_n).$$

where each R_i as a positive literal, $x_1, \ldots, x_n$ are sets of variables, and y is a set of free variables such that $y \subseteq \bigcup_i x_i$. We assume that the remaining variables $\bigcup_i x_i \setminus y$ are existentially quantified. The relation $q(y)$ thus denotes the answer of the query.

Example 2.1. *Consider the database schema of a university containing the relational symbols* $\mathsf{exams}(\text{cid}, \text{student}, \text{grade})$*,* $\mathsf{enrolled}(\text{student}, \text{program})$*, and* $\mathsf{mandatory}(\text{program}, \text{cid})$*. Assuming that a grade equals to zero is equivalent to a failure and abbreviating attribute names, the query asking for the students who failed any mandatory course in the program they are enrolled in can be expressed as follows*

$$q(\mathit{stud}) \leftarrow \mathit{exams}(\mathit{cid}, \mathit{stud}, \mathit{0}), \mathit{enrolled}(\mathit{stud}, \mathit{prog}), \mathit{mandatory}(\mathit{prog}, \mathit{cid}).$$

This simple fragment is found in many applications as it is equivalent to SELECT-PROJECT-JOIN queries in SQL [5] and Basic Graph Patterns in SPARQL [116]. For a more detailed study of CQs we refer to Abiteboul, Vianu, and Hull [5].

A *constraint satisfaction problem (CSP)* P is a triple $\langle V, D, C_t \rangle$, where V is a set of variables, D is a set of values, and C_t is a set of constraints. A constraint $(s_i, r_i) \in C_t$ consists of a tuple of variables s_i and a constraint relation r_i containing valid combinations of values for the variables s_i. A solution of P is a mapping $\phi \colon V \rightarrow D$, such that for each $(s_i, r_i) \in C_t$ the variables in s_i are mapped to a legal combination of values in r_i. For further discussion of CSPs we refer the reader to [42].

Example 2.2. *Consider a crossword puzzle such as the one in Figure 2.1, where each contiguous horizontal or vertical line of white cells has to be filled with a word chosen from a given set of words. Understandably, the words chosen for any pair of intersecting lines must have the same letter in the right position. This puzzle can be represented as a CSP $P = \langle V, D, C_t \rangle$. Each cell of the puzzle is a variable in V, which is defined as $V = \{a, b, c, d, e, f, g, h, i, j, k, l\}$. For simplicity, we assume that the domain D only consists of letters of the alphabet. Given a relation of words W with characters in D, the set C_t of constraints contains a constraint c_i for each contiguous horizontal or vertical line of white cells that can be filled with appropriate words in W. In this case, the constraint c_1 defined over the variables $w_1 = \langle a, b, c \rangle$ can take values in $r_1 = W$. If $W = \{\langle d, o, g \rangle, \langle g, o, d \rangle, \langle o, d, d \rangle\}$ as in Figure 2.1b, then the assignment $\langle a, b, c, d, e, f, g, h, i, j, k, l \rangle = \langle \mathrm{g, o, d, o, o, d, o, g, d, o, d, d} \rangle$ is a solution.*

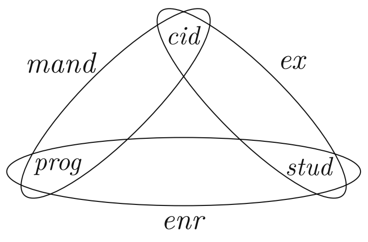
A *hypergraph* $H = (V(H), E(H))$ is a pair consisting of a set of vertices $V(H)$ and a set of non-empty (hyper)edges $E(H) \subseteq 2^{V(H)}$. We assume w.l.o.g. that there are no isolated vertices, i.e., for each $v \in V(H)$, there is at least one edge $e \in E(H)$ such that $v \in e$. We will often use H to denote the set of edges $E(H)$, with the understanding that $V(H) = \bigcup_{e \in E(H)} e$. A *subhypergraph* H' of H is then simply a subset of (the edges of) H. Given $U \subseteq V(H)$ the *induced subhypergraph of H w.r.t. U* is the hypergraph $H[U]$ s.t. $V(H[U]) = U$ and $E(H[U]) = \{e \cap U \mid e \in E(H)\} \setminus \{\emptyset\}$. The degree of a hypergraph H is the maximum number of incident edges on any vertex, formally, $\max_{v \in V(H)} |\{e \in E(H) \mid v \in e\}|$. The rank of a hypergraph H is the size of the largest edge of H, i.e., $\max_{e \in E(H)} |e|$.

Given a formula Q representing a CQ, the hypergraph H_Q corresponding to Q has $V(H_Q) = \mathrm{vars}(Q)$ and, for each atom a of Q, $\mathrm{vars}(a) \in E(H_Q)$. On the other hand, given a CSP $P = \langle V, D, C_t \rangle$, the hypergraph H_P of P is defined with $V(H_P) = V$ and $E(H_P) = \{s_i \mid (s_i, r_i) \in C_t\}$.

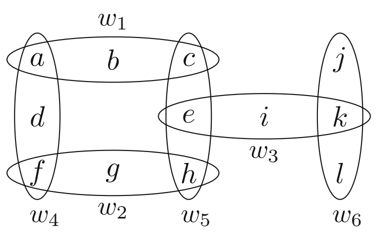
We are frequently dealing with sets of sets of vertices (e.g., sets of edges). For $S \subseteq 2^{V(H)}$, we write $\bigcup S$ and $\bigcap S$ as a short-hand for taking the union or intersection, respectively, of this set of sets of vertices, i.e., for $S = \{s_1, \ldots, s_\ell\}$, we have $\bigcup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

Example 2.3. *The hypergraph representing the query of Example 2.1 is shown in Figure 2.2a. It is worth noting that, since the attribute* grade *of the relation* exams *is bound to be 0 in this query, the edge corresponding to the relation* exams *has only arity 2.*

The hypergraph of the CSP of Example 2.2 is shown in Figure 2.2b. The set of vertices of each hypergraph is the set of variables of the corresponding CSP, while the set of edges matches the set of constraint scopes of the related CSP.



(a) Hypergraph H_Q from Example 2.1.



(b) Hypergraph H_P from Example 2.2.

Figure 2.2: The hypergraphs corresponding to the CQ of Example 2.1 and the CSP of Example 2.2.

2.2 Hypergraph Decompositions and Widths

We use $B(E)$ to denote the set of vertices of H *covered* by a certain set of edges E of H. More precisely, given a hypergraph $H = (V(H), E(H))$ and a set of edges $E \subseteq E(H)$, we define $B(E) = \bigcup_{e \in E} e$ as the set of all vertices of H contained in the set of edges E.

A *generalized hypertree decomposition* (GHD) [72] of a hypergraph $H = (V(H), E(H))$ is a tuple $\langle T, (B_u)_{u \in T}, (\lambda_u)_{u \in T} \rangle$ where $T = (N(T), E(T))$ is a tree, every B_u is a subset of $V(H)$, every λ_u is a subset of $E(H)$, and the following hold:

(1) For every edge $e \in E(H)$, there is a node u in T, such that $e \subseteq B_u$, and

(2) for every vertex $v \in V(H)$, $\{u \in T \mid v \in B_u\}$ is a connected subtree in T, and

(3) for each $u \in T$, $B_u \subseteq B(\lambda_u)$ holds.

By slight abuse of notation, we write $u \in T$ to express that u is a node in $N(T)$. We refer to the vertex sets B_u as the *bags* of the GHD, while we call the edge sets λ_u *edge covers* because $B_u \subseteq B(\lambda_u)$. Condition (2) is also called the *connectedness condition.*

We use the following notational conventions. 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 the decomposition. For a node $u \in T$, we write T_u to denote the subtree of T rooted at u. By slight abuse of notation, we will often write $u' \in T_u$ to denote that u' is a node in the subtree T_u of T. Finally, we also use $B(T_u) = \bigcup_{u' \in T_u} B_{u'}$.

The *width* of a GHD is defined as the largest size of any set λ_u over all nodes $u \in T$. The generalized hypertree width of H ($ghw(H)$) is the minimum width over all GHDs of H.

Example 2.4. *A GHD for the hypergraph H_P of Figure 2.2b is shown in Figure 2.3. It is easy to check that conditions (1)-(3) are satisfied. Since $\max_{u \in T} |\lambda_u| = 2$, this GHD has width 2. It is also possible to prove that $ghw(H_P) = 2$. Indeed, the hypergraph H_P has a cycle and, thus, it cannot have width 1.*

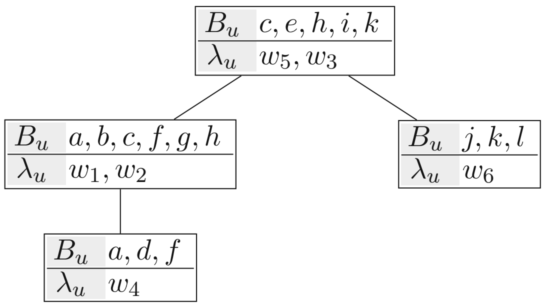


Figure 2.3: A GHD of width 2 for the hypergraph H_P of Figure 2.2b.

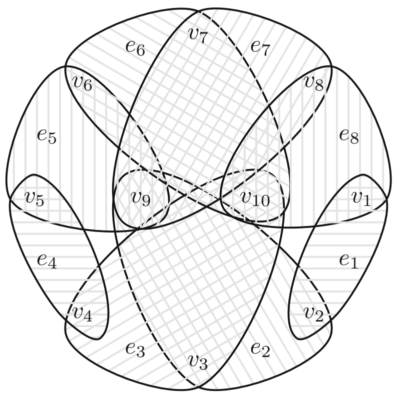


Figure 2.4: Hypergraph H_0 of Example 2.5 (taken from [7]).

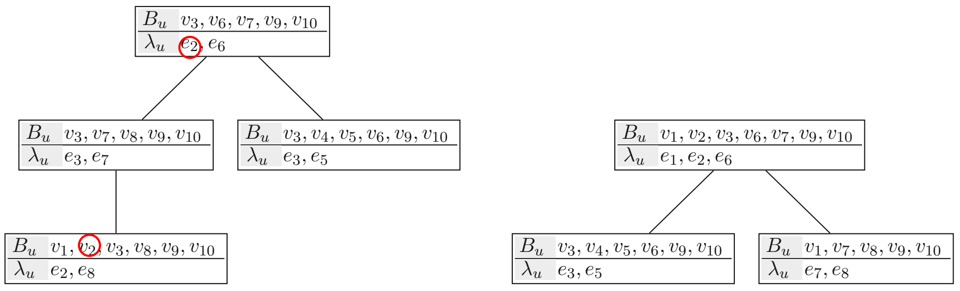
In this work we consider also another class of decompositions, which is formally a subset of all GHDs. A *hypertree decomposition* (HD) of H is a GHD satisfying the following additional condition (referred to as the "special condition" in [72]):

(4) For each $u \in T$, $B(T_u) \cap B(\lambda_u) \subseteq B_u$, where $B(T_u)$ denotes the union of all bags in the subtree of T rooted at u.

Because of condition (4), it is important to consider T as a *rooted* tree in case of HDs, while the root of T can be arbitrarily chosen or simply ignored in GHDs. Similarly to GHDs, the *width* of an HD is defined as the largest size of any set λ_u over all nodes $u \in T$. The hypertree width of H ($hw(H)$) is the minimum width over all HDs of H.

Some relationships between *ghw* and *hw* are known. For instance, $ghw(H) \leq hw(H)$, thus making GHDs better if we assume that smaller widths lead to faster solving times. It also known that $ghw(H) \leq 3 \cdot hw(H) + 1$, for any hypergraph H [7]. Then, from a theoretical point of view, *ghw* and *hw* are equivalent.

Example 2.5. *Consider the hypergraph H_0 of Figure 2.4 (this example is an adaptation of [7, 73], which was in turn inspired from [6]). We now examine two decompositions of H_0, which we show in Figure 2.5. While Figure 2.5a is a GHD, Figure 2.5b is*



B_u $v_3, v_6, v_7, v_9, v_{10}$
λ_u e_2, e_6
B_u $v_3, v_7, v_8, v_9, v_{10}$
λ_u e_3, e_7
B_u $v_3, v_4, v_5, v_6, v_9, v_{10}$
λ_u e_3, e_5
B_u $v_1, v_2, v_3, v_8, v_9, v_{10}$
λ_u e_2, e_8
B_u $v_1, v_2, v_3, v_6, v_7, v_9, v_{10}$
λ_u e_1, e_2, e_6
B_u $v_3, v_4, v_5, v_6, v_9, v_{10}$
λ_u e_3, e_5
B_u $v_1, v_7, v_8, v_9, v_{10}$
λ_u e_7, e_8

(a) A GHD of H_0 of width 2. (b) An HD of H_0 of width 3.

Figure 2.5: Comparison between a GHD and an HD of H_0

an HD. It is easy to check that the two decompositions satisfy Conditions (1)-(3), but only Figure 2.5b satisfies Condition (4). In fact, while Figure 2.5b is also a GHD (by definition), Figure 2.5a is not an HD. Indeed, the edge cover λ_r of the root r of Figure 2.5a contains the edge e_2, which in turn contains the vertex v_2. Nevertheless, this vertex does not appear in the bag of the root. While this is allowed in a GHD, it violates Condition (4) of HDs as v_2 could have been "covered" already in the root, but it appears only in a leaf of the decomposition. Moreover, the two decompositions have different widths. In fact, it can be proved that $ghw(H_0) = 2$ and $hw(H_0) = 3$.

Given a hypergraph $H = (V(H), E(H))$ and a GHD $D = \langle T, (B_u)_{u \in T}, (\lambda_u)_{u \in T} \rangle$ of H, where $T = (N(T), E(T))$, we say that D is *normal form* if, for each $r \in N(T)$, and for each child s of r, the following conditions hold:

(5) There is exactly one $[B_r]$-component C_r such that $B(T_s) = C_r \cup (B_s \cap B_r)$,

(6) $B_s \cap C_r \neq \emptyset$, where C_r is the $[B_r]$-component satisfying Condition (5),

(7) $B(\lambda_s) \cap B_r \subseteq B_s$.

This normal form prevents redundancies to appear in a decomposition, such as two adjacent nodes in T having identical bags and edge covers. Notably, it is always possible to transform any decomposition into normal form without changing the width.

Theorem 2.1 ([72])**.** *Given a GHD of width k of a hypergraph H, there exists a GHD of width k of H in normal form.*

In our work, we also discuss other forms of decompositions such as *tree decompositions* and *fractional hypertree decompositions.*

A *tree decomposition* (TD) [123, 122] of a hypergraph $H = (V(H), E(H))$ is a tuple $\langle T, (B_u)_{u \in T} \rangle$ where $T = (N(T), E(T))$ is a tree, every B_u is a subset of $V(H)$, and conditions (1) and (2) are satisfied. In essence, a tree decomposition is a GHD where edge covers are ignored. In fact, GHDs generalize TDs. The *width* of a TD is defined as $w - 1$, where w is the size of the largest bag B_u over all nodes $u \in T$. The tree width of H ($tw(H)$) is the minimum width over all TDs of H.

Consider a hypergraph $H = (V(H), E(H))$ and an *edge weight function* $\gamma\colon E(H) \to [0,1]$. We define the set $B(\gamma)$ of all vertices covered by γ and the weight of γ as

$$\begin{aligned} B(\gamma) &= \left\{ v \in V(H) \mid \sum_{e \in E(H),\, v \in e} \gamma(e) \geq 1 \right\}, \\ weight(\gamma) &= \sum_{e \in E(H)} \gamma(e). \end{aligned}$$

We call γ a *fractional edge cover* of a set $X \in V(H)$ with edges in $E(H)$ if $X \subseteq B(\gamma)$. We also consider an *integral edge cover* as a function $\lambda\colon E(H) \to \{0,1\}$, i.e., a fractional edge cover with values in $\{0,1\}$. Thus, an edge cover, or any set of edges, $\lambda \subseteq E(H)$ can be treated as an integral edge cover where $\lambda(e) = 1$, for each $e \in \lambda$, and the weight is the cardinality of λ. In the following, to emphasize the nature of the function we are dealing with, we will use γ for fractional edge covers and λ for integral edge covers.

A *fractional hypertree decomposition* (FHD) [79] of a hypergraph $H = (V(H), E(H))$ is a tuple $\langle T, (B_u)_{u \in T}, (\gamma_u)_{u \in T} \rangle$, such that $\langle T, (B_u)_{u \in T} \rangle$ is a TD of H and the following condition holds:

(3a) For each $u \in T$, $B_u \subseteq B(\gamma_u)$ holds, i.e., γ_u is a fractional edge cover of B_u.

Every GHD is an FHD where the edge covers λ are integral. Conversely, there exist FHDs that are not GHDs. Thus, FHDs are a generalization of GHDs. The *width* of an FHD is defined as the maximum weight of the functions γ_u over all nodes $u \in T$. The fractional tree width of H ($fhw(H)$) is the minimum width over all FHDs of H.

The following relationship is known for any hypergraph H:

$$fhw(H) \leq ghw(H) \leq hw(H) \leq tw(H)$$

It is also known that all of these widths are different, i.e., for each pair of widths (w_1, w_2) such that $w_1(H) \leq w_2(H)$ for any hypergraph H, there is a class of hypergraphs $\mathcal{H}$ such that w_1 is bounded on $\mathcal{H}$ while w_2 is not.

2.3 The Complexity of Computing Decompositions

We formally introduce the CHECK(w, k) problem of checking whether a given hypergraph H has a decomposition of width $w(H) \leq k$, for any fixed width function w and integer k. It is understood that in the case of CHECK(ghw, k) the output has to be a GHD, while in the case of CHECK(hw, k) we want to construct an HD.

Check(w, k)	
Instance:	hypergraph H
Output:	decomposition of H of width $w(H) \leq k$ if it exists, no otherwise

We have already mentioned that the Check(hw, k) problem is tractable for any fixed integer k [72]. On the contrary, the Check(ghw, k) problem is NP-complete for any $k \geq 2$ [53, 73]. However, there are some structural properties of hypergraphs that make the Check(ghw, k) problem tractable for large classes of hypergraphs. Here we refer to the terminology of [69] as it makes uniform the one originally introduced in [53].

Definition 2.1. *For $c \geq 1$, $d \geq 0$, a hypergraph $H = (V(H), E(H))$ is a (c, d)-*hypergraph *if the intersection of any c edges in $E(H)$ has at most d elements, i.e., for every subset $E' \subseteq E(H)$ with $|E'| = c$, we have $|\bigcap E'| \leq d$.*

Definition 2.2. *A hypergraph $H = (V(H), E(H))$ has* c-multi-intersection size d *if H is a (c, d)-hypergraph. In the special case of $c = 2$, we speak of intersection size of H, while if we do not have a particular c in mind, we simply speak of multi-intersection size of H.*

Definition 2.3. *A class $\mathcal{C}$ of hypergraphs satisfies the* bounded multi-intersection property (BMIP)*, if there exist $c \geq 1$ and $d \geq 0$, such that every H in $\mathcal{C}$ is a (c, d)-hypergraph. As a special case, $\mathcal{C}$ satisfies the* bounded intersection property (BIP)*, if there exists $d \geq 0$, such that every H in $\mathcal{C}$ is a $(2, d)$-hypergraph.*

Definition 2.4 ([136])**.** *Let $H = (V(H), E(H))$ be a hypergraph, and $X \subseteq V(H)$ a set of vertices. Denote by $E(H)|_X = \{X \cap e \mid e \in E(H)\}$. X is called* shattered *if $E(H)|_X = 2^X$. The* Vapnik-Chervonenkis dimension (VC dimension) *of H is the maximum cardinality of a shattered subset of V. We say that a class $\mathcal{C}$ of hypergraphs has bounded VC-dimension, if there exists $v \geq 1$, such that every hypergraph $H \in \mathcal{C}$ has VC-dimension $\leq v$.*

Note that a hypergraph H with degree bounded by δ is a $(\delta+1, 0)$-hypergraph. Similarly, a hypergraph H with rank bounded by ρ is a $(1, \rho)$-hypergraph. Thus, bounded degree as well as bounded rank implies the BMIP, which in turn implies bounded VC-dimension [53].

Example 2.6. *The hypergraph H_0 of Figure 2.4 has intersection size and 3-multi-intersection size equal to 1. For any $c \geq 4$, the c-multi-intersection size of H_0 is 0.*

The aforementioned properties help to solve the Check(ghw, k):

Theorem 2.2 ([53, 69])**.** *Let $\mathcal{C}$ be a class of hypergraphs. If $\mathcal{C}$ has the BMIP, then the* Check(ghw, k) *problem is solvable in polynomial time for any $k \geq 1$. Consequently, this tractability holds if $\mathcal{C}$ has bounded degree or the BIP (each of which implies the BMIP).*

Additionally, if $\mathcal{C}$ has bounded VC-dimension, then the fhw can be approximated in polynomial time up to a logarithmic factor.

Algorithm 2.1: A schematic top-down GHD algorithm.

```
Input: Hypergraph $H$
Output: A GHD of $H$ with width $\leq k$, or Nil if none exists
Parameter: Integer $k \geq 1$

Function FindDecomp($H'$: Hypergraph)
    for $S \in$ Separators($H, k$) do
        compute $U \subseteq V(H')$ such that $U \subseteq B(S)$
        children $\leftarrow \emptyset$
        for $H_c \in [S]$-components of $H'$ do
            $\mathcal{D} \leftarrow$ FindDecomp($H_c$)
            if $\mathcal{D}$ == Nil then
                continue outer loop
            end
            children $\leftarrow$ children $\cup \{\mathcal{D}\}$
        end
        return Hypertree($B_u \leftarrow U$, $\lambda_u \leftarrow S$, Children $\leftarrow$ children)
    end
    return Nil
end
begin /* Main */
    return FindDecomp($H$)
end
```

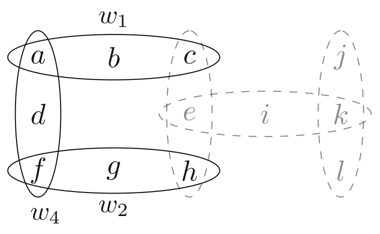
2.4 Top-Down Construction of GHDs

For a set $U \subseteq V(H)$ of vertices, we define $[U]$-components of a hypergraph H through the following steps.

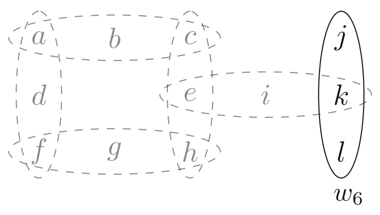
- We define $[U]$-*adjacency* as a binary relation on $E(H)$ as follows: two edges e_1 and e_2 are $[U]$-*adjacent*, if $(e_1 \cap e_2) \setminus U \neq \emptyset$ holds.
- We define $[U]$-*connectedness* as the transitive closure of the $[U]$-*adjacency* relation.
- A $[U]$-*component* of H is a maximally $[U]$-connected subset $C \subseteq E(H)$.

For a set of edges $S \subseteq E(H)$, we say that C is $[S]$-*connected* as a short-cut for C is $[U]$-*connected* with $U = \bigcup_{e \in S} e$. We also call S a *separator* and C an $[S]$-*component*. The *size of an* $[S]$-*component* C is defined as the number of edges in C.

Many top-down GHD algorithms such as [52, 74, 76] can be described by the same schema. Here we simply refer to a generic top-down schema with the understanding that each implementation presents its own peculiarities. A pseudo-code description of such a schematic top-down GHD algorithm is given in Algorithm 2.1.



(a) Component C_1.



(b) Component C_2.

Figure 2.6: The two components C_1, C_2 of Example 2.7 obtained by removing $\{w_5, w_3\}$ from H_P of Figure 2.2b.

For a fixed $k \geq 1$, Algorithm 2.1 takes as input a hypergraph H and either builds a GHD of H of width $\leq k$ or rejects if none can be found. The main algorithm is entirely based on the function `FindDecomp`, which is initially called on the input hypergraph H. Each call of `FindDecomp` is over a particular set of edges, referred to as H'. This recursive function forms the core of this schematic description. A recursive function is chosen for notational simplicity, a real implementation could also just use nested loops or other ways to implement iteration. The function `FindDecomp` iterates over all possible edge separators of size $\leq k$ with edges from the original input hypergraph H, as seen in line 2. Note that the particulars of how these edge separators are computed can vary greatly between specific implementations. Given an edge set S, the algorithm computes a suitable vertex set U such that $U \subseteq B(S)$. Again, the specific way in which this set is computed is implementation dependent. The two sets U and S will form the bag and the edge cover of the root node u of the GHD found for the subhypergraph H'.

Next, in line 5, the algorithm determines all $[S]$-components of H', which we denote as $C_1, \dots, C_\ell$. In order to be sure that this algorithm terminates, we assume that the produced GHDs are in normal form. They always exist due to Theorem 2.1. Hence, the algorithm recursively searches for a GHD of the subhypergraphs H_i with $E(H_i) = C_i$ and $V(H_i) = \bigcup C_i$. We can see this inside function `FindDecomp` in lines 5-11. If all recursive calls succeed, the function terminates by constructing a GHD with root u and subtrees covering the components C_i, seen in line 12.

Example 2.7. *We decompose the hypergraph H_P of Figure 2.2b so to obtain the GHD shown in Figure 2.3. We follow the generic top-down Algorithm 2.1, to which we refer here as $\mathcal{A}$. For $k = 2$, $\mathcal{A}$ takes as input H_P and computes a GHD of H_P of width 2, if it exists. Firstly, $\mathcal{A}$ guesses a separator of size ≤ 2 with edges in $C = E(H_P)$, which will be used as the edge cover λ_u for the root node u of the GHD. Suppose that $\lambda_u = \{w_5, w_3\}$ and $B_u = \{c, e, h, i, k\}$ is a suitable choice for the bag. Then the root u of the decomposition will be exactly the same of Figure 2.3. At this point, $\mathcal{A}$ computes the $[\lambda_u]$-components $C_1, C_2 \subseteq C$ with $C_1 = \{w_1, w_2, w_4\}$, $C_2 = \{w_6\}$ (Figure 2.6). Each C_i is now recursively decomposed. Since $|C_2| \leq 2$, it can be "covered" by a single node of the decomposition. On the contrary, $|C_1| > 2$, thus $\mathcal{A}$ starts a recursive call on C_1 and guesses the separator $S = \{w_1, w_2\}$ and computes the new bag $\{a, b, c, f, g, h\}$. As the component C_1 is split* *w.r.t.* S*, we obtain a single component* $C_3 \subseteq C_1$*, with* $C_3 = \{w_4\}$*. This component can be "covered" by a single node. Finally, all nodes are attached to their respective fathers except for the root. The resulting decomposition is returned.*

2.5 Summary

In this chapter we recapitulated the basic definitions and the main results that we will need throughout the rest of this thesis. We gave formal definitions for CQs, CSPs and hypergraphs. We also showed how to transform instances of this kind into hypergraphs. Then, we defined HDs and GHDs and the corresponding measures of width hw and ghw. While each HD is also a GHD, the contrary does not hold and we recalled one example (hypergraph H_0) where $ghw(H_0) < hw(H_0)$. It is thus profitable to compute GHDs over HDs. Nevertheless, computing a GHD of width $\leq k$ is harder than computing an HD of width $\leq k$. We thus summarized which properties of hypergraphs make the first problem tractable. Finally, we sketched a generic top-down algorithm that is at the base of most implementations for computing (G)HDs. This will help us understand the algorithms presented later in this work.

CHAPTER 3

The Computation of GHDs through Balanced Separators

In the previous chapter, we recalled the main definitions of hypergraphs and structural decomposition methods. We introduced two kinds of decomposition, namely, *hypertree decompositions (HDs)* and *generalized hypertree decompositions (GHDs)*, and discussed their quirks and differences. The main advantage of HDs over GHDs is that an HD of width $\leq k$ can be computed in polynomial time for any fixed integer k [72], whereas GHDs cannot be efficiently computed for any $k \geq 2$, in general [73, 53, 69]. On the other hand, for any hypergraph H, $ghw(H) \leq hw(H)$, meaning that algorithms answering CQs or solving CSPs by exploiting low-width decompositions might not fully enjoy the benefits brought by this kind of techniques.

Consequently, searching tractable fragments of GHD computation is still an active field. Already in [53], Fischl, Gottlob, and Pichler identified numerous tractable cases of GHD computation characterized by some intersection properties of hypergraphs. In particular, they proved that the Check(ghw, k) problem is tractable for all classes of hypergraphs enjoying the *bounded multi-intersection property*, which we defined in Chapter 2. Based on this result, Fischl et al. proposed in [50] some fixed-parameter tractable algorithms that compute GHDs efficiently for quite a few hypergraphs by exploiting the properties defined above.

Notably, Fischl proposes in [49] a novel method to characterize "no"-instances of the Check(ghw, k) problem. This algorithm, referred to in [49] as `BalSep`, is based on the so-called *balanced separators*, which are a well-known concept in graph theory generalized to hypergraphs. This procedure was used in [49] as an empirical test to determine whether a hypergraph allows for a GHD of width $\leq k$. However, neither a general algorithm to compute GHDs nor a formal proof of correctness had not been provided there. An algorithm based on this procedure has been delineated only in [50].

This chapter is devoted to the computation of GHDs for classes of hypergraphs having bounded intersection size. Beginning with a summary of the efficient GHD algorithms based on hypergraph intersection properties introduced in [50, 49], we provide a solid background on the tractable cases of GHD computation. We thus turn our attention to the `BalSep` procedure informally sketched in previous work. We turn `BalSep` into a complete algorithm for computing GHDs in polynomial time by combining the ideas of [50] regarding exploiting intersection size for efficient computation of GHDs with the testing procedure in [49] utilizing balanced separators. In this algorithm, a hypergraph H is decomposed by splitting H into components smaller than half the size of H. These are recursively decomposed according to the same criterion until a base case is reached. This way of constructing a GHD diverges slightly but significantly from the general schema shown in Algorithm 2.1. Thus, it is not trivial to see whether the resulting algorithm is correct. Therefore, we introduce the necessary formalism to prove that the revised `BalSep` algorithm is sound and complete.

This chapter is structured into four sections. We initially recapitulate in Section 3.1 previous relevant work related to the computation of GHDs in tractable cases such as the one of bounded intersection size. Section 3.2 reports on our extension of the `BalSep` procedure delineated in [49] for computing a GHD. Here we also introduce the necessary formalism which is needed to guarantee the correctness of this algorithm. We then proceed in Section 3.3 with the proof of soundness and completeness of `BalSep`. We summarize the results achieved in this chapter in Section 3.4.

The contents of this chapter are based on the papers published together with Wolfgang Fischl, Georg Gottlob, and Reinhard Pichler [52]. Section 3.1 summarizes the main contributions of Wolfgang Fischl [49].

3.1 Tractable Cases of GHD Computation

In this section we recapitulate some of the main contributions of Wolfgang Fischl [49], on which the author of this thesis does not claim any ownership. It is however important to present here the essential points of Fischl's work as these will find full application in our development of the algorithm based on *balanced separators* in Section 3.2.

Firstly, we provide the necessary theoretical background for the computation of GHDs for classes of hypergraphs satisfying the *Bounded Intersection Property (BIP)*. As mentioned in Section 2, in this case the problem of computing a GHD of width $\leq k$ is tractable, when k is fixed. Building on this first subsection, we present two implementations of the general algorithm for computing GHDs in case of BIP, namely `GlobalBIP` and `LocalBIP`. These variants differ in the time of the computation of certain sets of edges which are added to the input hypergraph to make the computation efficient.

3.1.1 Theoretical Background

Even though the CHECK(ghw, k) problem was proved to be intractable for any $k \geq 2$ in [53], Fischl, Gottlob, and Pichler identified tractable fragments of the problem characterized by the bounded (multi-)intersection property (Definition 2.3). Indeed, according to Theorem 2.2, classes of hypergraphs having bounded (multi-)intersection size allow for GHD computation in polynomial time. In the following we focus on the case of bounded intersection size, which is the maximum size of any intersection of two edges of a hypergraph, and explain how this property leads to polynomial-time computation of GHDs. The general theoretical algorithm will be simply called *ghw-algorithm*.

The ghw-algorithm consists of two phases. Given a hypergraph $H = (V(H), E(H))$, it first computes a set $f(H, k)$ of new edges obtained by intersecting certain combinations of edges from $E(H)$ which have non-empty intersection. This set of edges is thus added to the original hypergraph H. It is important that $f(H, k)$ can be computed in polynomial time. Moreover, the set $f(H, k)$ has the property that $ghw(H) = k$ if and only if $hw(H') = k$, where $H' = (V(H), E(H) \cup f(H, k))$. Thus, it is possible to use CHECK(hw, k) to solve CHECK(ghw, k) and achieve tractability. This is indeed the second step of the ghw-algorithm. An HD of H' of width $\leq k$ is computed in polynomial time. If the resulting HD makes use of any edge $e' \in f(H, k)$ in any edge cover, this is substituted by some edge $e \in E(H)$ such that $e \supseteq e'$. The result is a GHD of H of width $\leq k$.

The set of edges $f(H, k)$ is defined as follows.

$$f(H, k) = \bigcup_{e \in E(H)} \left(\bigcup_{e_1, \ldots, e_j \in (E(H) \setminus \{e\}),\, j \leq k} 2^{(e \cap (e_1 \cup \cdots \cup e_j))} \right) \tag{3.1}$$

Informally, for each $e \in E(H)$, all subsets of intersections of e with up to k edges of H different from e are computed. Although $f(H, k)$ could in general contain an exponential number of elements, for fixed k and intersection size of H bounded by d, the set $e \cap (e_1 \cup \cdots \cup e_j)$ contains at most $d \cdot k$ elements and, therefore, $|f(H, k)|$ is polynomially bounded.

Since the ghw-algorithm relies on the computation of HDs, a new version of the hw-algorithm in [76], called `NewDetKDecomp`, has been made available. This program contains also the implementations of all the algorithms in this section, but we defer a detailed discussion of the complete library to Chapter 4.

3.1.2 The **`GlobalBIP`** Algorithm

Given a hypergraph $H = (V(H), E(H))$ and an integer $k \geq 1$, a straightforward implementation of the ghw-algorithm consists in initially computing the set $f(H, k)$, creating the hypergraph $H' = (V(H), E(H) \cup f(H, k))$, and finally computing an HD of H' of width $\leq k$, if it exists. Since here the set $f(H, k)$ is computed a priori *globally* for the whole hypergraph, this algorithm is called `GlobalBIP`.

Algorithm 3.1: The `GlobalBIP` Algorithm.

```
Input: Hypergraph $H$
Output: A GHD of $H$ with width $\leq k$, or Nil if none exists
Parameter: Integer $k \geq 1$
1  begin
2      compute $f(H, k)$ as in Equation 3.1
3      $H' \leftarrow (V(H), E(H) \cup f(H, k))$
4      $\mathcal{D} \leftarrow$ ComputeHD($H', k$)
5      if $\mathcal{D} ==$ Nil then
6          return Nil
7      end
8      foreach $u \in T$ of $\mathcal{D}$ do
9          foreach $e' \in ((\lambda_u \cap f(H, k)) \setminus E(H))$ do
10             choose $e \in E(H)$ such that $e \supseteq e'$
11             $\lambda_u \leftarrow (\lambda_u \setminus \{e'\}) \cup \{e\}$
12         end
13     end
14     return $HD$
15 end
```

Algorithm 3.1 is a detailed description of how `GlobalBIP` works. The input consists of a hypergraph H and a constant k, while the output is a GHD of H of width $\leq k$, if it exists, and **Nil** otherwise. In line 2, the computation $f(H, k)$ as per Equation 3.1 is performed and then, in line 3, the hypergraph H' is created. This is obtained by adding the subedges in $f(H, k)$ to H. In line 4, an external routine computing HDs is called on H' and k as a black box and its output is stored in the variable $\mathcal{D}$. If $\mathcal{D}$ is **Nil**, a GHD of width $\leq k$ of H does not exist, therefore **Nil** is returned. Otherwise,the decomposition needs to be "fixed" as described in lines 8-13. In particular, for each node u of $\mathcal{D}$, and for each edge of $f(H, k)$ in λ_u, i.e., $e' \in (\lambda_u \cap f(H, k))$ that is not an edge in the original hypergraph H, the edge e' is substituted with an edge $e \in E(H)$ such that $e \supseteq e'$. In this way we obtain a new edge cover λ'_u such that $B_u \subseteq B(\lambda_u) \subseteq B(\lambda'_u)$, but also $|\lambda_u| = |\lambda'_u|$. Thus, the new decomposition still satisfies all the properties of a GHD and its width is still $\leq k$. Eventually, in line 14, $\mathcal{D}$ is returned.

3.1.3 The `LocalBIP` Algorithm

The main drawback of `GlobalBIP` is that the size of $f(H, k)$, though polynomial, could be huge even for relatively small-sized hypergraphs. On the other hand, not every edge in $f(H, k)$ is necessary at any given step of the GHD computation and some of them might not be useful at all. Therefore, while adding all of the edges $f(H, k)$ a priori is not always a viable solution, restricting the computation only to those edges that might be actually necessary in a given algorithmic step might be beneficial to reduce the number of edges to add to H. The approach used follows from an observation about the role played by $f(H,k)$ in the tractability proof in [53].

Recall that the proof uses $\textsc{Check}(hw,k)$ on the hypergraph H' to answer $\textsc{Check}(ghw,k)$ for the hypergraph H. To do this in a sound way, the set $f(H,k)$ has to contain all the edges that could be used to cover possible bags of H' in an HD without changing the width. Consider a top-down construction of a GHD of H. At some point we might want to choose, for some node u, a bag B_u such that $v \notin B_u$ for some variable $v \in B(\lambda_u) \cap V(T_u)$. This choice would violate condition (4) of HDs and would not be allowed for the computation of an HD. In particular, there is an edge e with $v \in e$ and $e \in \lambda_u$. For this reason, the set $f(H,k)$ contains an edge e' such that $e' \subset e$ and $v \notin e'$. Hence, we can substitute e with e' in the cover λ_u (i.e., $e \notin \lambda_u$ and $e' \in \lambda_u$) to eliminate the violation of condition (4). Moreover, because of the connectedness condition, there is no need to look at the intersection of e with arbitrary edges in $E(H)$, instead we consider only the intersections of e with unions of edges that may possibly occur in bags of T_u. In other words, for each node u of the decomposition, only an appropriate subset $f_u(H,k) \subseteq f(H,k)$ is considered. More specifically, for the current node u, let $H_u \subseteq H$ be the component to be decomposed. Then, $f_u(H,k)$ is defined as follows:

$$f_u(H,k) = \bigcup_{e \in E(H)} \left(\bigcup_{e_1,\ldots,e_j \in (E(H_u) \setminus \{e\}),\, j \leq k} 2^{(e \cap (e_1 \cup \cdots \cup e_j))} \right) \tag{3.2}$$

The resulting algorithm is called `LocalBIP` because the set of edges $f_u(H,k)$ is computed *locally* for each node u during the construction of the decomposition. This algorithms follows closely the description of Algorithm 2.1, but it differs in the search of the separators in line 2. In particular, while decomposing H, the algorithm first tries all possible ℓ-combinations of edges in $E(H)$ (with $\ell \leq k$) and only if the search does not succeed, it tries ℓ-combinations of subedges in $f_u(H,k)$.

3.2 A GHD Algorithm based on Balanced Separators

So far we have presented two adaptations of the theoretical *ghw*-algorithm from [53]. While `GlobalBIP` and `LocalBIP` can be seen as an extension of `NewDetKDecomp` to compute GHDs and, at the same time, exploiting bounded intersection size to achieve tractability, they do not really introduce any significant algorithmic innovation. In the following we describe a method for computing GHDs that makes use of particular sets of edges called *balanced separators*. We first extend the terminology of Chapter 2 for our purposes, then give a detailed description of the `BalSep` algorithm, and finally prove that our algorithm is sound and complete.

3.2.1 Balanced Separators and Special Edges

Recall that a *hypergraph* is a pair $H = (V(H), E(H))$ consisting of a set $V(H)$ of *vertices* and a set $E(H)$ of hyperedges (or, simply *edges*), which are non-empty subsets of $V(H)$.

Since we assume that hypergraphs do not have isolated vertices, we can identify a hypergraph H with its set of edges $E(H)$. Then, a subhypergraph H' of H is a subset of (the edges of) H.

Starting off with a hypergraph H, the `BalSep` algorithm has to deal with subhypergraphs $H' \subseteq H$ augmented by a set S_p of special edges. A special edge is simply a set of vertices from H. Intuitively, special edges correspond to bags B_u in a GHD of H. Thus, an *extended subhypergraph* of H is of the form $H' \cup S_p$, where $H' \subseteq H$ is a subhypergraph and S_p is a set of special edges.

We now extend three crucial definitions from hypergraphs to extended subhypergraphs, namely components, balanced separators and GHDs. We recall that, even though a separator is a set of vertices, it can be defined as a set of edges. Then, for $S \subseteq E(H)$, an $[S]$-component is a $[W]$-component with $W = \bigcup_{e \in S} e$. In our algorithm we use the fact that it is always possible to choose a separator λ_u such that $B(\lambda_u) = B_u$. This is due to the fact that the edges $f(H,k)$ are considered part of the input hypergraph [53]. Hence, there will not be any need to distinguish between vertex and edge separators. We start with components.

Definition 3.1 (Components of Extended Subhypergraphs)**.** *For a set $U \subseteq V(H)$ of vertices, we define $[U]$-components of an extended subhypergraph $H' \cup S_p$ of H as follows:*

- *We define* $[U]$-adjacency *as a binary relation on $H' \cup S_p$ as follows: two (possibly special) edges $f_1, f_2 \in H' \cup S_p$ 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 $H' \cup S_p$ is a maximally $[U]$-connected subset $C \subseteq H' \cup S_p$.*

Hence, if $C_1, \ldots, C_\ell$ are the $[U]$-components of $H' \cup S_p$, then $H' \cup S_p$ is partitioned into $C_0 \cup C_1 \cup \cdots \cup C_\ell$, such that $C_0 = \{f \in H' \cup S_p \mid f \subseteq U\}$.

We next define balanced separators. While we give a definition with respect to sets of vertices $U \subseteq V(H)$, they can be alternatively defined in terms of sets of edges $S \subseteq E(H)$ for the reason mentioned above.

Definition 3.2 (Balanced Separators)**.** *Let $H' \cup S_p$ be an extended subhypergraph of a hypergraph H and let $U \subseteq V(H)$ be a set of vertices of H. The set U is a* balanced separator *of $H' \cup S_p$ if for each $[U]$-component C_i of $H' \cup S_p$, $|C_i| \leq \frac{|H' \cup S_p|}{2}$ holds. In other words, no $[U]$-component must contain more than half the edges of $H' \cup S_p$.*

Finally, we extend GHDs to extended subhypergraphs.

Definition 3.3 (GHDs of Extended Subhypergraphs)**.** *Let H be a hypergraph and $H' \cup S_p$ an extended subhypergraph of H. A GHD of $H' \cup S_p$ is a tuple $\langle T, (B_u)_{u \in T}, (\lambda_u)_{u \in T} \rangle$, where $T = (N(T), E(T))$ is a tree, and B_u and λ_u are labeling functions, which map to* *each node $u \in T$ two sets, $B_u \subseteq V(H)$ and $\lambda_u \subseteq E(H) \cup S_p$. For a node u, we call B_u the* bag *and λ_u the* edge cover *of u. The set $B(\lambda_u)$ of vertices "covered" by λ_u is defined as $B(\lambda_u) = \{v \in V(H) \mid v \in f, f \in \lambda_u\}$. The functions λ_u and B_u have to satisfy the following conditions:*

1. *For each node $u \in T$, either*
   a) $\lambda_u \subseteq E(H)$ and $B_u \subseteq B(\lambda_u)$, or
   b) $\lambda_u = \{s\}$ for some $s \in S_p$ and $B_u = s$.
2. *If, for some $u \in T$, $\lambda_u = \{s\}$ for some $s \in S_p$, then u is a leaf node.*
3. *For each $e \in H' \cup S_p$, there is a node $u \in T$ such that $e \subseteq B_u$.*
4. *For each vertex $v \in V(H)$, $\{u \in T \mid v \in B_u\}$ is a connected subtree of T.*

The width of a GHD *is defined as* $\max\{|\lambda_u| : u \in T\}$.

Clearly, also H itself is an extended subhypergraph of H with $H' = H$ and $S_p = \emptyset$. It is readily verified that the above definition of GHD of an extended subhypergraph $H' \cup S_p$ and the definition of GHD of a hypergraph H coincide for the special case of taking H as an extended subhypergraph of itself.

In [72], a normal form of hypertree decompositions was introduced. We will carry the notion of normal form over to GHDs of extended subhypergraph. To this end, it is convenient to first define the set of edges *exclusively* covered by some subtree of a GHD:

Definition 3.4. *Let $H' \cup S_p$ be an extended subhypergraph of some hypergraph H and $\mathcal{D} = \langle T, (B_u)_{u \in T}, (\lambda_u)_{u \in T}\rangle$ a GHD for $H' \cup S_p$. For a node $u \in T$, we write T_u to denote the subtree of T rooted at u. Moreover, we define the set of edges exclusively covered by T_u as $exCov(T_u) = \{f \in H' \cup S_p \mid \exists v \in T_u : f \subseteq B_v\}$.*

Our normal form of GHDs is then defined as follows:

Definition 3.5 (GHD of Extended Subhypergraphs Normal Form)**.** *We say that a GHD $\langle T, (B_u)_{u \in T}, (\lambda_u)_{u \in T}\rangle$ of an extended subhypergraph $H' \cup S_p$ is in normal form, if for the root node r of T, the following property holds:*

> *Let $u_1, \dots, u_\ell$ be the child nodes of r in T and let $T_{u_1}, \dots, T_{u_\ell}$ denote the subtrees in T rooted at $u_1, \dots, u_\ell$, respectively. Then $exCov(T_{u_1}), \dots, exCov(T_{u_\ell})$ are precisely the $[B(\lambda_r)]$-components of $H' \cup S_p$.*

Intuitively, each subtree T_i below the root "covers" the edges of precisely one *$[B_r]$-component of $H' \cup S_p$.*

The following lemma is an immediate extension of previous results for hypergraphs to extended subhypergraphs.

Lemma 3.1. *Let $H' \cup S_p$ be an extended subhypergraph of some hypergraph H and suppose that there exists a GHD $\mathcal{D}$ of width $\leq k$ for $H' \cup S_p$. Then there also exists a GHD $\mathcal{D}'$ in normal form of width $\leq k$ for $H' \cup S_p$, such that B_r is a balanced separator of $H' \cup S_p$ for the root node r of $\mathcal{D}'$.*

Proof. The lemma combines two results from [72] and [7], respectively.

Our normal form relaxes the normal form of HDs introduced in Definition 5.1 in [72]. The transformation into normal form can be taken over almost literally from the proof of Theorem 5.4 in [72] for establishing the normal form of HDs.

The existence of a balanced separator as the root of a GHD is implicit in the definition of "hyperlinkedness" and Theorem 19 in [7]. Again, it can be easily taken over to our case of an extended subhypergraph (i.e., to take also special edges into account). Actually, it can also be easily proved directly by starting off at the root r of an arbitrary GHD of $H' \cup S_p$ in normal form and checking if all the components covered by the subtrees below have size at most $\frac{|H' \cup S_p|}{2}$. If so, we already have the desired form. If not, there must be one subtree $T_{r'}$ rooted at a child r' of r, such that $exCov(T_{r'})$ is greater than $\frac{|H' \cup S_p|}{2}$. We then apply the normal form transformation also to $T_{r'}$ and check recursively if all the components covered by the subtrees below all have size at most $\frac{|H' \cup S_p|}{2}$. By repeating this recursive step, we will eventually reach a node u, such that B_u is a balanced separator. Then we simply take this node as the root and again apply the normal form transformation of the proof of Theorem 5.4 in [72] to this new root node and the subtrees immediately below it. □

3.2.2 The `BalSep` Algorithm

The `BalSep` algorithm, which computes GHDs through balanced separators, is presented in Algorithm 3.2. For a fixed integer $k \geq 1$, the procedure takes as input a hypergraph H and computes a GHD of H of width $\leq k$ if it exists, otherwise **Nil** is returned. The main body of the algorithm consists of a simple call to the Decompose function in line 35 with parameters H and an empty set of special edges. The result of this function is thus returned after the recursion has been performed.

This recursive function constitutes the core of the algorithm and, given as input a hypergraph H' and a set of special edges S_p, computes a GHD of $H' \cup S_p$ of width $\leq k$ if it exists. Lines 2-11 deal with the two base cases of the algorithm. If $H' \cup S_p$ has only one edge, we create a decomposition made of a single node u whose edge cover λ_u contains the only edge of $H' \cup S_p$ and the bag $B_u = V(H' \cup S_p)$, i.e., all the vertices of the extended subhypergraph, which are exactly the ones covered by λ_u. We deal with the case of $E(H' \cup S_p)$ having two edges, where at most one is a special edge, in a similar way. Here we create a decomposition composed of two nodes u, v, each of which "covers" a single edge of $H' \cup S_p$. The node v is then attached as a child of u and returned. Note that, being a GHD undirected, the choice of the root is irrelevant.

If the extended subhypergraph has at least three edges, we have to decompose it until we reach one of the two base cases. In line 12, we initialize the object *BalSepIt*, which is an iterator over the balanced separators of size $\leq k$ of $H' \cup S_p$ with edges in H. The iterator *BalSepIt* produces, one by one, all the ℓ-combinations of edges in H, for each $\ell \leq k$, to find a balanced separator for $H' \cup S_p$. Moreover, if all the combinations of full edges fail, the function uses subedges of H to generate separators corresponding to elements of the set $f(H, k)$ as per Equation 3.1.

In the while loop of lines 13-30, we recursively decompose $H' \cup S_p$. We start by fixing the edge cover and the bag for the current node u of the GHD. Then, we take the next balanced separator of $H' \cup S_p$ as λ_u and compute $B_u = B(\lambda_u)$ for the reasons discussed above. Moreover, initially the node u has no children. At this point we want to compute a GHD for each $[B_u]$-component of $H' \cup S_p$. These components have to be decomposed in such a way that it will be possible to attach each decomposition of theirs to the current node u as children without violating any condition of Definition 3.3. We describe this process in a separate function, called by the Decompose function in line 17.

Algorithm 3.3 computes the set of extended subhypergraphs corresponding to $[B_u]$-components of $H' \cup S_p$ and introduces, in each of them, a new special edge B_u to keep a connection to $H' \cup S_p$ and, ultimately, to the other nodes in the decomposition. We assume here the existence of a function ConnectedComponents which, given a hypergraph H, computes the connected components of H in the standard way. First, Algorithm 3.3 computes the hypergraph $H_u = (V_u, E_u)$ resulting from the removal of all vertices in B_u from $H' \cup S_p$. Then, in lines 5-12, for each connected component of H_u, a new subhypergraph of $H' \cup S_p$ is created. The new subhypergraph is stored in a variable c, which is a pair consisting of a hypergraph H and a set S_p of special edges. For a single subhypergraph, the new set of special edges $c.S_p$ is composed of the edges of S_p intersecting the current component *comp* plus a new special edge $s = B_u$ corresponding to the separator B_u. We can then construct the hypergraph $c.H$ as follows. Its set of edges E contains the edges of H' intersecting the current component *comp*, while its set of vertices is the union of E and $c.S_p$. We then add c to the set of results $\mathcal{C}$, which we finally return in line 13.

Back in the Decompose function, in lines 17-25, we recursively compute a GHD for each extended subhypergraph returned by Algorithm 3.3. If the decomposition $\mathcal{D}$ returned in line 18 is not **Nil**, we add it to the set *subDecomps* of the children of the current node u, otherwise, we set *subDecomps* to **Nil** and break the loop. This means that a GHD of width $\leq k$ of the current subhypergraph could not be found. At the end of the loop, we check whether *subDecomps* itself is **Nil**. If this is the case, it means that one of the recursive calls of the Decompose function was unsuccessful. We then have to continue the while loop of lines 13-30 and try the next balanced separator. Nevertheless, in case all the recursive calls of the Decompose function were successful, Algorithm 3.4 builds the resulting GHD and returns it in line 29. If the algorithm exhausts all the choices of balanced separators for $H' \cup S_p$ and exits the while loop of lines 13-30, it means that it is impossible to create a GHD of width $\leq k$ for $H' \cup S_p$ and we return **Nil** in line 31.

Finally, we describe Algorithm 3.4 that, given a bag B_u, an edge cover λ_u, and a set *children* of GHDs, returns a GHD with root u and children *children*. This function has the important task to reroot the subdecompositions before attaching them to the current node u. Without rerooting the resulting GHD could possibly be wrong. We start off creating the node u with labels B_u and λ_u. Then, for each child $\mathcal{D} \in$ *children*, let T be the tree structure of $\mathcal{D}$. We identify the node $\hat{r} \in T$ having $B_{\hat{r}} = B_u$ and designate it as the new root of T. Now, for each child $c_{\hat{r}}$ of $\hat{r}$, we attach $c_{\hat{r}}$ as a child of u. In other words, we attach every subtree rooted at a child node of $\hat{r}$ to u. In the end, we return the resulting decomposition.

3.3 A Proof of Correctness for `BalSep`

In this section we prove that Algorithm 3.2 is sound and complete. The correctness of Algorithm 3.3 and Algorithm 3.4 is immediate and thus omitted.

Theorem 3.1. *Let H be a hypergraph and $k \geq 1$ an integer. Algorithm 3.2 called on H with parameter k returns a GHD of H of width $\leq k$ if and only if $ghw(H) \leq k$.*

We prove the soundness and the completeness of Algorithm 3.2 separately. Nevertheless, we want to point out that the main procedure of the algorithm consists of a call to Decompose function with input H and an empty set of special edges. Thus, in the next proofs, we will actually prove that Decompose function called on (H', S_p) with parameter k returns a GHD of $H' \cup S_p$ of width $\leq k$ if and only if $ghw(H' \cup S_p) \leq k$ with respect to Definition 3.3. Note that in case of a hypergraph H and a set of special edges $S_p = \emptyset$, Definition 3.3 coincides with the usual definition of GHD.

Proof. (Soundness) We show that if the Decompose function called on (H', S_p) with parameter k returns a GHD of $H' \cup S_p$ of width $\leq k$, then such a decomposition actually exists and $ghw(H' \cup S_p) \leq k$. We proceed by induction over the size of $H' \cup S_p$, i.e., $|E(H' \cup S_p)|$. For the base case, we assume $|H' \cup S_p| \leq 2$. In case $|H' \cup S_p| = 1$, we return a GHD made of a single node whose edge cover λ consists of the only edge $H' \cup S_p$, which also cover all of the vertices of the hypergraph. Such a decomposition has width 1 and clearly satisfies all the conditions of Definition 3.3. In case $|H' \cup S_p| = 2$, we create two nodes u, v, each one corresponding to an edge of $H' \cup S_p$ and we attach v as a child of u. Note that both u and v are leaf nodes. It is easy to verify that also in this case we return a valid GHD of $H' \cup S_p$.

For the induction step, suppose that the recursive function Decompose correctly returns a GHD of $H' \cup S_p$ of width $\leq k$ for each $H' \cup S_p$ such that $|H' \cup S_p| \leq j$, for some $j \geq 2$. Now suppose that $|H' \cup S_p| = j + 1 \geq 3$ and that Decompose(H', S_p) returns a GHD of $H' \cup S_p$ of width $\leq k$. We have to show that then there indeed exists such a decomposition of $H' \cup S_p$.

In this case, Algorithm 3.2 only returns a GHD in line 29. The program successfully reaches this line only if the following happens:

- In line 14, a balanced separator λ_u of $H' \cup S_p$ is chosen.
- In line 17, the extended subhypergraphs of $H' \cup S_p$ with respect to $B_u = B(\lambda_u)$ are computed; each extended subhypergraph corresponds to a $[B_u]$-component of $H' \cup S_p$ plus a new special edge $s = B_u$.
- For each extended subhypergraph $c.H \cup c.S_p$, the recursive call to Decompose function in line 18 is successful, i.e., it returns a GHD of $c.H \cup c.S_p$ of width $\leq k$.

We are assuming that $|H' \cup S_p| \geq 3$ and that λ_u is a balanced separator of $H' \cup S_p$. Let $C_1, \ldots, C_\ell$ be the ℓ $[B_u]$-components of $H' \cup S_p$. All extended subhypergraphs $C_i \cup \{B_u\}$ are strictly smaller than j. Hence, by the induction hypothesis, for each $i \in \{1, \ldots, \ell\}$, there indeed exists a GHD $\mathcal{D}_i = \langle T_i, (B_{i,u})_{u \in T}, (\lambda_{i,u})_{u \in T} \rangle$ of width $\leq k$ for the extended subhypergraph $C_i \cup \{B_u\}$.

It is left to show that Algorithm 3.4 correctly constructs a GHD of width $\leq k$ of $H' \cup S_p$. For each $\mathcal{D}_i$, let $\hat{r}_i$ be the node of T_i with $B_{i,\hat{r}_i} = B_u$ and $\lambda_{i,\hat{r}_i} = \{B_u\}$. By construction, we know that the node $\hat{r}_i$ exists in every $\mathcal{D}_i$, it is always a leaf, and it has the same bag and edge cover λ in every decomposition. Let $T_i = (N(T_i), E(T_i))$ be the tree structure of $\mathcal{D}_i$ and, w.l.o.g., assume that the node sets $N(T_i)$ are pairwise disjoint. We define the tree structure $T = (N(T), E(T))$ and the functions B_u and λ_u of $\mathcal{D}$ as follows:

- $N = (N(T_1) \setminus \{\hat{r}_1\}) \cup \cdots \cup (N(T_\ell) \setminus \{\hat{r}_\ell\}) \cup \{r\}$, where r is a new (root) node.
- For the definition of $E(T)$ recall that each $\hat{r}_i$ is a leaf node in its decomposition. Let e_i denote the edge between $\hat{r}_i$ and its parent. Then we define $E(T)$ as $E(T) = (E(T_1) \setminus \{e_1\}) \cup \cdots \cup (E(T_\ell) \setminus \{e_\ell\}) \cup R$ with $R = \{[r, \hat{r}_1], \ldots, [r, \hat{r}_\ell]\}$.
- $\lambda_r = \lambda_u$ and $B_r = B_u$.
- For every $v \in N \setminus \{r\}$, there exists exactly one i, such that $v \in N(T_i)$. We set $\lambda_v = \lambda_{i,v}$ and $B_v = B_{i,v}$.

Intuitively, the GHD $\mathcal{D}$ is obtained by taking in each GHD $\mathcal{D}_i$ the node $\hat{r}_i$ as the root node and combining all GHDs $\mathcal{D}_i$ into a single GHD by merging their root nodes to a single node r. This is possible since all nodes $\hat{r}_i$ have the same edge cover λ and bags. It is easy to verify that the resulting GHD is indeed a GHD of width $\leq k$ of the extended subhypergraph $H' \cup Sp$. ☐

Proof. (Completeness) For any hypergraph H' and set of special edges S_p, we prove that if $ghw(H' \cup S_p) \leq k$, then the Decompose function on input (H', S_p) returns a GHD of width $\leq k$ of $H' \cup S_p$. Again, we proceed by induction on $|H' \cup S_p|$.

The base case of $|H' \cup S_p| \leq 2$ is dealt with in lines 2-11 of Algorithm 3.2. In this case, we simply construct a GHD of width 1 for $|H' \cup S_p|$ and return it.

For the induction step, suppose $|H' \cup S_p| \leq j$, for some $j \geq 2$, and $ghw(H' \cup S_p) \leq k$. Then, the Decompose function on input (H', S_p) returns a GHD of width $\leq k$ of $H' \cup S_p$. Now assume that $|H' \cup S_p| = j + 1 \geq 3$ and that $ghw(H' \cup S_p) \leq k$. We have to show that the Decompose function on input (H', S_p) returns a GHD of width $\leq k$ of $H' \cup S_p$.

By Lemma 3.1, we may assume, w.l.o.g., that the GHD $\mathcal{D} = \langle T, (B_u)_{u \in T}, (\lambda_u)_{u \in T} \rangle$ is in normal form and that B_r is a balanced separator of $H' \cup S_p$ for the root node r of $\mathcal{D}$. Let $S \subseteq E(H)$ denote the edge cover λ of r, i.e., $\lambda_r = S$.

When the Decompose function is called on input (H', S_p), the while loop in lines 13-30 eventually generates all possible balanced separators of size $\leq k$ of $H' \cup S_p$, unless it returns on line 29 before the end of the loop. Remember that the object *BalSepIt* not only generates edge separators with edges in $E(H)$, i.e., the original hypergraph on which Algorithm 3.2 is called, but it also uses edges in $f(H, k)$, i.e., subedges of edges in $E(H)$. Thus, at some point, in line 14, we will choose the separator $\lambda_u = S$, equivalently, $B_u = B_r$.

Let $C_1, \dots, C_\ell$ denote the $[B_u]$-components of $H' \cup S$. Since $\mathcal{D}$ is in normal form, we know that the root node r has ℓ child nodes such that $C_i = exCov(T_i)$, where T_i is the subtree in T rooted at n_i for $i \in \{1, \dots, \ell\}$. Recall from Definition 3.4 that we write $exCov(T_i)$ to denote the set of edges exclusively covered by T_i.

Now consider the extended subhypergraph $C_i \cup \{B_u\}$, for an arbitrary $i \in \{1, \dots, \ell\}$. Since B_r is a balanced separator of $H' \cup S_p$, we have that $|C_i \cup \{B_u\}| \leq j$. Moreover, there exists a GHD $\mathcal{D}_i$ of $C_i \cup \{B_u\}$, namely the subtree of $\mathcal{D}$ induced by the nodes in T_i plus r. Hence, by the induction hypothesis, calling the Decompose function with the input corresponding to the extended subhypergraph $C_i \cup \{B_u\}$ returns a valid GHD. This means that, in our call of the function Decompose with input (H', Sp), we have the following behavior:

- On line 18, the Decompose function is called recursively for all extended subhypergraphs $C_i \cup \{B_u\}$.
- Each call of the Decompose function returns a GHD for each respective extended subhypergraph.
- The results of these recursive calls are collected in line 20 in the variable *subDecomps*.
- Hence, after exiting the loop of lines 17-25, the return statement in line 29 is executed.

The call to Algorithm 3.4 correctly produces the desired GHD as discussed in the soundness proof. Finally, the Decompose function indeed returns a GHD of width $\leq k$ of $H' \cup S_p$. □

We end this section with a note on the complexity of Algorithm 3.2. Since the set of edge separators explored in `BalSep` correspond to the set of edges $f(H, k)$ of Equation 3.1, it follows that, for any fixed $k \geq 1$, `BalSep` is a fixed-parameter tractable algorithm with respect to the intersection size of the input hypergraph H. Indeed, it has been proven in [53] that in the case of BIP, the size of $f(H, k)$ is bounded by $m^{k+1} \cdot 2^{k \cdot i}$, where m is the number of edges of H and i is the intersection size of H as defined in Definition 2.2.

3.4 Summary

In this chapter, we dealt with the problem of computing GHDs in tractable cases. We offered a review of the most significant properties of classes of hypergraphs allowing for fixed-parameter tractable computation of GHDs and described some algorithms based on these ideas. We learned how `GlobalBIP` and `LocalBIP` exploit bounded intersection size to compute GHDs efficiently.

Furthermore, we built on the ideas developed in [50, 49] to extend the simple `BalSep` procedure for checking "no"-instances of the CHECK(ghw, k) problem to a complete algorithm for constructing GHDs in polynomial time. Because of the elaborate way in which `BalSep` constructs a GHD, we developed a theory based on the concept of extended subhypergraphs, which allows us to compute GHDs when some "special edges" are introduced into a hypergraph during the decomposition procedure.

Finally, we concluded the chapter by proving the correctness of the revised `BalSep` algorithm. We added a powerful instrument to our toolkit for decomposing hypergraphs. The benefits derived from this algorithm will be evident in the next chapter, where we will demonstrate the capabilities of `BalSep` in decomposing hypergraphs stemming from real-world problems.

Algorithm 3.2: The `BalSep` Algorithm.

Input: Hypergraph H
Output: A GHD of H with width $\leq k$, or **Nil** if none exists
Parameter: Integer $k \geq 1$

```
Function Decompose($H'$: Hypergraph, $S_p$: Set of special edges)
    if $|E(H' \cup S_p)| == 1$ then
        return Hypertree($B_u \leftarrow V(H' \cup S_p)$, $\lambda_u \leftarrow E(H' \cup S_p)$)
    end
    if $(|E(H' \cup S_p)| == 2) \wedge (|S_p| \leq 1)$ then
        Let $e_1, e_2$ be the two edges of $H' \cup S_p$
        $u \leftarrow$ Hypertree($B_u \leftarrow e_1$, $\lambda_u \leftarrow \{e_1\}$)
        $v \leftarrow$ Hypertree($B_v \leftarrow e_2$, $\lambda_v \leftarrow \{e_2\}$)
        $u$.Children $\leftarrow \{v\}$
        return $u$;
    end
    $BalSepIt \leftarrow$ InitBalSepIterator($H, H', S_p, k$)
    while HasNext($BalSepIt$) do
        $\lambda_u \leftarrow$ Next($BalSepIt$)
        $B_u \leftarrow B(\lambda_u)$
        $subDecomps \leftarrow \emptyset$
        foreach $c \in$ Separate($H', S_p, B_u$) do
            $\mathcal{D} \leftarrow$ Decompose($c.H, c.S_p$)
            if $\mathcal{D} \neq$ Nil then
                $subDecomps \leftarrow subDecomps \cup \{\mathcal{D}\}$
            else
                $subDecomps \leftarrow$ Nil
                break
            end
        end
        if $subDecomps ==$ Nil then
            continue
        end
        return BuildGHD($B_u, \lambda_u, subDecomps$)
    end
    return Nil
end
begin /* Main */
    Make $H$ and $k$ globally visible
    return Decompose($H$, $\emptyset$)
end
```

Algorithm 3.3: The `Separate` Function used by `BalSep`.

Input: Hypergraph H', Set of special edges S_p, Set of vertices B_u.
Output: The set of subhypergraphs of $H' \cup S_p$ w.r.t. B_u.

```
1  begin
2      $V_u \leftarrow V(H') \setminus B_u$
3      $E_u \leftarrow \{e \cap V_u \mid e \in E(H' \cup S_p)\}$
4      $\mathcal{C} \leftarrow \emptyset$
5      foreach $comp \in$ ConnectedComponents($V_u, E_u$) do
6          $c \leftarrow$ initialize pair ($H = \textbf{Nil}$, $S_p = \textbf{Nil}$)
7          $c.S_p \leftarrow \{s \in S_p \mid s \cap comp \neq \emptyset\} \cup \{B_u\}$
8          $E \leftarrow \{e \in E(H') \mid e \cap comp \neq \emptyset\}$
9          $V \leftarrow V(E) \cup V(c.S_p)$
10         $c.H \leftarrow (V, E)$
11         $\mathcal{C} \leftarrow \mathcal{C} \cup \{c\}$
12     end
13     return $\mathcal{C}$
14 end
```

Algorithm 3.4: The `BuildGHD` Function used by `BalSep`.

Input: Set of vertices B_u, Set of edges λ_u, Set of GHDs *children*.
Output: A GHD with root u and children *children*.

```
1  begin
2      $u \leftarrow$ Hypertree($B_u \leftarrow B_u$, $\lambda_u \leftarrow \lambda_u$)
3      foreach $\mathcal{D} \in children$ do
4          Let $T$ be the tree structure of $\mathcal{D}$
5          $\hat{r} \leftarrow$ Reroot($T, B_u$)
6          foreach $c_{\hat{r}} \in \hat{r}$.Children do
7              $u$.Children $\leftarrow u$.Children $\cup \{c_{\hat{r}}\}$
8          end
9      end
10     return $u$
11 end
```

CHAPTER 4

Benchmarking Decomposition Algorithms

In Chapter 3, we presented our novel `BalSep` algorithm, which, given a hypergraph H and an integer k, computes a GHD of H of width $\leq k$ in polynomial time, if it exists. Even though this is a powerful instrument in our toolbox of hypergraph decomposition algorithms, the performance of `BalSep` is highly influenced by the width and the intersection size of the input hypergraph, which are the crucial parameters involved in the time complexity of the algorithm. It is therefore seen as desirable for these two numbers to be low. Nevertheless, this limitation is exclusively justified if the hypergraphs stemming from real-world CQs and CSPs have these characteristics.

Fischl et al. gave a convincing affirmative answer with their work on *HyperBench* [50]. This benchmark comprises thousands of hypergraphs deriving from as many CQs and CSPs for assessing the characteristics of real hypergraphs and the capability of decomposition algorithms to decompose them. The analysis of these hypergraphs revealed that most have low width and intersection sizes, thus providing a compelling argument in favor of decomposition algorithms. On the other hand, the benchmark composition is slightly unbalanced towards CSPs. Indeed, out of the 3070 hypergraphs collected by Fischl et al., only 1035 originated from CQs, which is just about a third of the total. Giving a closer look at these instances, we also realized that, except for the 70 SPARQL queries obtained from Bonifati et al. [27], the rest are all SQL queries, some of which preprocessed to adhere to a "flat" and simplified subset of the language corresponding to pure conjunctive queries. Consequently, we decided to improve the work initiated in [50] by providing a more representative and inclusive view of the query-answering instances landscape.

On the one hand, we wanted to solve the problem of the under-representation of SPARQL queries. Therefore, we looked at the literature for query sources and found out that an immense amount of SPARQL queries originating from Wikidata had been released by Malyshev et al. [104] and had been successively studied by Bonifati et al. in [28]. These queries are particularly interesting because of their provenance. Indeed, they include human-generated queries and "synthetic" ones generated by algorithms, further categorized into queries for which an answer was provided within a given timeout and queries that timed out. In a pioneering effort, Bonifati et al. analyzed the over 200 million queries in this repository by examining their structural properties, including hypertree width. Their study revealed that 273974 unique queries have $hw \geq 2$. However, other hypergraph structural properties such as *ghw* and the various intersection sizes have not been examined there.

On the other hand, we wanted to enlarge the set of SQL queries by including queries from different applications and having a complex structure. In this case, we turned our attention to the TPC benchmarks, which are well-established in the database community. While the TPC-H benchmark [132] has been included in [50], the TPC-DS benchmark [131], which contains a variety of industry-relevant queries to test general purpose decision support systems, is missing there. This benchmark contains 113 queries with a complex structure that does not immediately correspond to a linear form of a SELECT-PROJECT-JOIN conjunctive query. Furthermore, the tools provided in [50] cannot parse them. Nevertheless, most of these queries can be converted into a purely conjunctive form. When this is impossible, we can still analyze their conjunctive parts separately.

The benchmarking of efficient decomposition algorithms on complex queries is the theme of this chapter. We begin our study by clearly distinguishing between simple and complex SQL queries. The latter intuitively correspond to those queries using views, nested subqueries, and other features of the SQL language "hiding" the conjunctive nature of a query. Being able to distinguish these two classes of queries, we define an algorithm to transform complex SQL queries into a collection of simple ones. Indeed, while some queries are irreducible in a purely conjunctive form, it is still possible to analyze their conjunctive parts separately. In this case, we split the original query into maximal conjunctive components, which we then transformed into simple queries and, eventually, hypergraphs. The resulting algorithm, referred to as *hg-tools*, has been implemented and made publicly available at `https://github.com/dmlongo/hgtools`. We then analyze the TPC-DS benchmark and the Wikidata SPARQL queries from [104, 28]. In analogy with the original work on HyperBench [50], our goal is twofold: while we want to test the hypothesis that the various GHD algorithms described in the previous chapter are still adequate for the new hypergraphs coming from complex SQL queries and SPARQL queries, we also benchmark their performances. We conclude that this is indeed the case: even complex SQL queries as well as SPARQL queries typically have low width.

This chapter proceeds with four sections. First, we present our methodology to translate complex SQL queries into hypergraphs in Section 4.1. Here we also give additional information about the translation of CSPs. Next, in Section 4.2, we discuss the new hypergraphs from the TPC-DS benchmark and the Wikidata SPARQL queries and the integration of these new instances into the old dataset. Then, Section 4.3 reports on the experiments performed on the integrated dataset. Here, we analyze the structural properties of the hypergraphs we collected and test the efficient GHD algorithms exploiting intersection size. Finally, Section 4.4 summarizes the results presented in this chapter.

The contents of this chapter are based on the article [52] published together with Wolfgang Fischl, Georg Gottlob, and Reinhard Pichler. Moreover, related content about the challenges of benchmarking decomposition algorithms appeared in [66]. While the translation strategy proposed in Section 4.1 and the successive collection and translation of the TPC-DS queries using this method are an original contribution of the author of this thesis, a note on the contents of Section 4.2 and Section 4.3 is necessary. These two sections present experiments carried out on a version of HyperBench which integrates the original version presented in [50] with the new instances described above. Since similar experiments had been already carried out by Wolfgang Fischl in [49] on the original version of HyperBench, we do not claim any ownership over the design of the experiments nor on the collection and translation of the instances already present in [49]. On this note, we also acknowledge that the hypergraphs of the Wikidata SPARQL queries have been kindly provided to us by Angela Bonifati, Wim Martens, and Thomas Timm. Nevertheless, both the original HyperBench instances from [50] and the hypergraphs of the Wikidata SPARQL queries from [28] have undergone some preprocessing before being integrated. In particular, where applicable, the old queries from [50] have been retranslated with our new parser *hg-tools*. This operation led to the discovery of some mistakes in the previous translation of the TPC-H queries, which we fixed in the new version of HyperBench. Moreover, we filtered the hypergraphs of the 273947 unique SPARQL queries from [28] by removing duplicates on the hypergraph level. Consequently, we recognized that $99,87\%$ of the instances were duplicates, which we did not integrate into the new dataset. This procedure left us with 354 unique hypergraphs. We provide a detailed account of these processes in Section 4.2. Finally, the reiteration of the experiments in [49, 50] carried out in Section 4.3 on the new dataset constitutes a “test of replicability” of the results in [49, 50] that, at the same time, strengthens them. Indeed, we reach the same conclusion of [49, 50] even on a more varied dataset.

4.1 Translation of CQs and CSPs into Hypergraphs

Since all decomposition algorithms work on hypergraphs, the translation of CQs and CSPs into this representation is essential. However, there are many languages that are used to define instances of these problems and each of them requires a specific translation algorithm. In this section, we focus on the SQL language for queries and the XCSP3 language for constraint satisfaction problems. While motivating the choices made in designing the translation process, we describe a new methodology for translating complex SQL queries. The algorithms described here have been implemented and made publicly available at `https://github.com/dmlongo/hgtools`.

4.1.1 From Simple to Complex SQL Queries

The SQL language comes with a varied set of different features, some of which are even system dependent. This offers users many ways to express the same query, but it is a problematic aspect when it comes to the translation of a query into a hypergraph. As seen in Chapter 2, only pure conjunctive queries can be transformed into hypergraphs and this form of CQs corresponds to basic SELECT-PROJECT-JOIN queries in SQL. Therefore, we need to differentiate between *simple* queries, which can be immediately translated into hypergraphs, and *complex* queries, which require some manipulation before they can be translated.

We say that an SQL query is *simple* if it is a SELECT-FROM-WHERE statement in which the WHERE clause is only allowed to be a conjunction of join conditions, i.e., equality conditions between columns of different tables. More explicitly, such a query must not contain any comparison, negation, disjunction, or subqueries, such as nested SELECT statements, neither in the FROM nor in the WHERE clauses.

Example 4.1. *Recall the excerpt of a university schema with relations* exams(cid, student, grade)*,* enrolled(student, program)*, and* mandatory(program, cid) *of Example 2.1. The query asking for the students who took at least one exam in any mandatory course of the program they are enrolled in is naturally stated in SQL as follows.*

```
SELECT exams.student
FROM   exams, enrolled, mandatory
WHERE  exams.cid = mandatory.cid
   AND exams.student = enrolled.student
   AND enrolled.program = mandatory.program;
```

It is easy to verify that this is a simple query and it can be straightforwardly translated into a hypergraph (details will follow).

In our efforts to develop an algorithm that transforms more general forms of SQL queries into hypergraphs, we prioritize preserving the fundamental structure of the query. While this approach may require us to neglect certain features that are necessary for answering the query, we believe it is a reasonable trade-off. As a result, we introduce the concept of a *quasi simple* query, which is an SQL query that does not contain nested subqueries and can be simplified by removing any number of conditions in the WHERE clause.

Moreover, we point out that the SELECT clause of a query is not relevant in constructing its hypergraph. Therefore, we allow the substitution of the original SELECT clause with the alternative "SELECT *" clause, which denotes the list of all attributes of tables in the FROM clause. Once again, we emphasize that this step simplifies the study of more general queries at the minor cost of changing the actual answers to the original query, which is for our purposes, anyway, irrelevant.

Example 4.2. *Consider the following modification of the query of Example 4.1 over the same university schema.*

```
SELECT exams.student, exams.cid
FROM   exams, enrolled, mandatory
WHERE  exams.cid = mandatory.cid
  AND  exams.student = enrolled.student
  AND  enrolled.program = mandatory.program
  AND  enrolled.program = 'Psychology'
  AND  exams.grade ≠ 0;
```

This query asks for the pairs (s, e) of students and exams such that s has passed e (grade $\neq 0$) and e is mandatory for the Psychology program, where s is enrolled. This query is quasi-simple in that it becomes simple by removing the last two conditions.

Queries containing nested subqueries deserve a special treatment. In SQL a subquery is defined via a nested SELECT statement. Although the presence of subqueries automatically makes the whole query non-conjunctive, their inner structure might still be of interest and, hence, it can be extracted and analyzed separately. We call *complex* SQL queries all those queries containing any number of subqueries. However, whereas some subqueries can be evaluated separately from the outer query, there are some that cannot. In order to distinguish between the two, we classify subqueries into two types: simple subqueries, which can generate their results independently of the statement they are embedded in, and correlated subqueries, which need data from their outer query in order to be executed. In our translation algorithm, we discard correlated subqueries and retain simple ones. These will be then extracted from the outer query and the two of them analyzed separately.

Example 4.3. *For every exam, the following SQL query asks for the students, along with course and grade information, who achieved the highest grade in that exam.*

```
SELECT e.student, e.cid, e.grade
FROM   exams e
WHERE  e.grade = (
       SELECT MAX(grade)
       FROM   exams e2
       WHERE  e2.cid = e.cid
);
```

This formulation contains a correlated subquery which has the task of finding the maximum grade for the exam currently inspected by the outer query. This reference from the subquery to the instance of the exams *in the outer query makes the independent evaluation of the subquery impossible.*

On the other hand, the following SQL query asking for the names of all programs that do not have the course with id 'CS101' as a mandatory course contains a simple subquery that can be executed separately from the outer query.

```
SELECT  program
FROM    enrolled
WHERE   program NOT IN (
        SELECT  program
        FROM    mandatory
        WHERE   cid = 'CS101'
);
```

Depending on the kind of subquery and on the part of the query in which it appears, we propose different treatments:

- If a query is of the form $q_1 \circ \cdots \circ q_n$, where each q_i is a query and $\circ \in \{\cup, \cap, \setminus\}$, we extract the single subqueries q_i and process them separately.
- If a subquery appears in the FROM clause, we convert it into a logical view.
- If a subquery contains a reference to an outer query, it must be discarded.

Regarding the second point, the presence of logical views in a query does not necessarily make it complex. This depends on the content of the view. Therefore, when a query contains a view, we expand it into the main body of the query with the standard methods.

4.1.2 Extracting Simple Queries from Complex Ones

When examining an SQL query containing subqueries, we are faced with two options: we either extract them or integrate them into the main body. Depending of the kind of subquery, the integration is not always possible and, in this case, we must extract the subquery and analyze it separately. In order to do it in a way that preserves the structure of the original query as much as possible, we build a graph representing dependencies between subqueries. At the end of the process, we extract queries which are independent and eliminate those which are mutually dependent.

We say that a subquery s_1 *depends* on a subquery s_2 if the result of s_1 can be computed only after computing the result of s_2. The dependency graph of a query Q is a directed graph $G = (S, D)$ where the set S contains the nodes corresponding to subqueries of Q and $(s_1, s_2) \in D$ is an arrow, for $s_1, s_2 \in S$, if s_1 depends on s_2. Given a query Q, we create its dependency graph G as follows:

1. Create a node $q \in S$ representing the outer query.
2. For each nested query s_i of q, create a new node $s_i \in S$ and an edge $(q, s_i) \in D$.

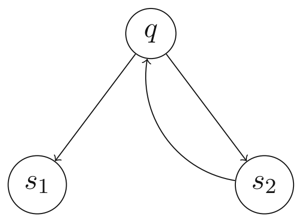


Figure 4.1: Dependency graph of the query in Example 4.4.

3. If s_i references a table defined in any ancestor s_j, create an edge (s_i, s_j).

4. Recursively examine s_i for nested queries.

Once we have built the graph, we identify the nodes which are involved in cycles and eliminate them. In particular, we consider q as a root and navigate the graph. Whenever we find a node having an edge pointing at an ancestor, we eliminate it together with all of its incoming and outgoing edges. Eventually, we end up with a forest in which we extract a query from each node.

Example 4.4. *The following query retrieves, for each exam that is not mandatory in the Computer Science program, all students, along with course and grade information, who achieved the highest grade in that exam.*

```
SELECT e.student, e.cid, e.grade
FROM   exams e
WHERE  e.cid NOT IN (
         SELECT cid
         FROM mandatory
         WHERE program ≠ 'Computer␣Science')
     AND  e.grade = (
         SELECT MAX(grade)
         FROM   exams e2
         WHERE  e2.cid = e.cid);
```

The dependency graph G of this query is shown in Figure 4.1. The node q, which is the root of G, represents the outer query. The nodes s_1 and s_2 in G represent the first and second subquery, respectively. As the result of the outer query can be computed only after computing the results of the subqueries, the two edges (q, s_1) and (q, s_2) are in G. Since s_2 references the table t_1 defined in q, G contains also the edge (s_2, q).

After the creation, we look for cycles in the graph. In this case, we see that there is no way to evaluate s_2 independently from q. Then, we remove s_2 and all of its incident edges. Finally, we extract a simple query from each node of the remaining graph.

4.1.3 Converting Simple Queries into Hypergraphs

Once we have extracted and simplified subqueries, we are left with simple SQL queries of the type

$$\textbf{SELECT } r_{i_1}.A_{j_1}, \dots, r_{i_z}.A_{j_z} \textbf{ FROM } r_1, \dots, r_m \textbf{ WHERE } \mathit{cond} \tag{4.1}$$

such that *cond* is a conjunction of conditions of the form $r_i.A = r_j.B$ or $r_i.A = c$, where c is a constant. Such queries are equivalent to conjunctive queries, thus it is easy to draw a connection to a CQ and transform it into a hypergraph. Nevertheless, in our case it makes more sense to go directly from an SQL query to the hypergraph.

Let Q be an SQL query of the form (4.1), then the hypergraph $H_Q = (V(H_Q), E(H_Q))$ corresponding to Q is obtained as follows. We first build the hypergraph induced by the FROM clause. Consider a relation $r(A_1, \dots, A_\ell)$ in the FROM clause. For each attribute A_i of r, we create a vertex $v_{A_i} \in V(H_Q)$. Then, we create the edge $r = \{v_{A_1}, \dots, v_{A_\ell}\} \in E(H_Q)$. Now, we modify the hypergraph according to the conditions in the WHERE clause. Let *cond* be such a condition. It can be of two forms:

- If *cond* is of the form $r_i.A = r_j.B$, we merge vertices v_A and v_b and modify their incident edges. W.l.o.g. assume v_A itself becomes the merged vertex. For each edge $r \in \{e \in E(H_Q) \mid v_B \in e\}$, we remove r from $E(H_Q)$ and add a new edge $r' = (r \setminus \{v_B\}) \cup \{v_A\}$.
- If *cond* is of the form $r_i.A = c$, with c constant, we remove v_A from $V(H_Q)$ and, for each edge $r \in \{e \in E(H_Q) \mid v_A \in e\}$, we remove r from $E(H_Q)$ and add a new edge $r' = r \setminus \{v_A\}$.

At the end of this procedure, we eliminate empty edges and multiple edges. Since SELECT clauses do not contribute to the query structure, we simply ignore them.

4.1.4 From CSPs to Hypergraphs

An important part of our benchmark consists of instances of Constraint Satisfaction Problems. The collected set presents different characteristics with respect to the ones found in CQs, thus their analysis offers a more varied picture of the hypergraphs encountered in applications. For this reason, the collected CSP instances have a significant practical aspect.

Most of the CSPs come from the XCSP [15] website. XCSP3 is an XML-based format used to represent constraint satisfaction problems. The language offers a wide variety of options to represent the most common constraints and frameworks, making it a solid intermediate format between different solvers. The collected XCSP instances are also used as a benchmark in solver competitions that are periodically organized.

From XCSP, there have been selected a total of 1953 instances with less than 100 extensional constraints such that all constraints are extensional, i.e., they are provided in the form of a relation. The choice on the number of constraints allows us to have an adequate number of instances of increasing difficulty. Moreover, algorithms which use GHDs to solve CSPs typically need the constraints in a relational form. Therefore, we picked only CSPs such that all constraints are extensional. These instances are divided into CSPs from concrete applications, called *CSP Application* in the sequel (1090 instances), and randomly generated CSPs, called *CSP Random* below (863 instances).

The instances fetched from the website are written in well-structured XML files in which variables and constraints are explicitly defined through the use of specific XML tags. The transformation of these instances into hypergraphs did not require a specific methodology since the authors of the XCSP3 format provide an extensive library for parsing the instances where most of the process is already automatized. Obviously, we still had to convert the object in memory into a hypergraph. To this end, we have reimplemented the behavior of some callback methods in such a way that, whenever the program reads a variable, it adds a vertex to the hypergraph, and, whenever it reads a constraint, it adds an edge containing the vertices corresponding to the variables affected by the constraint.

Our collection of CSPs also includes a third class, which we call *CSP Other*. These instances have been used in previous hypertree width analyses available at `https://www.dbai.tuwien.ac.at/proj/hypertree`. This set contains interesting examples coming from industry and a variety of different test examples [57]. In particular, a part of the hypergraphs is obtained from Daimler Chrysler and represents circuits and systems. A second part is a hypergraph translation of the circuits belonging to the well-known benchmark library of the IEEE International Symposium on Circuits and Systems (ISCAS). Finally, some hypergraphs correspond to grids extracted from pebbling problems. Since the instances are provided already as hypergraphs, no additional processing was necessary to incorporate them.

4.2 Integration of Complex Queries into Hyperbench

In this section, we describe the new version of *HyperBench*, which is composed of a benchmark and a web tool. Here we introduce the system and test environment used for the experiments and describe the CQs and CSPs in the benchmark.

4.2.1 System and Test Environment

Our system is composed of two libraries. A C++ library with the implementations of the GHD algorithms described in Chapter 3 was originally develop by Wolfgang Fischl for the work in [49] and successively improved by the author of this thesis. The code is available at `https://github.com/dmlongo/newdetkdecomp`. This library contains an extended version of the original `DetKDecomp` algorithm from [76], which is now called `NewDetKDecomp`. While the underlying *hw* algorithm is still the one from [76], the new implementation makes use of modern C++ constructs such as smart pointers for better memory management and maintainability. The code itself has also been optimized in several parts, thus improving overall performance w.r.t. the previous release. The software now also goes beyond the computation of HDs and it has been expanded to compute GHDs through the algorithms presented in Chapter 3. Moreover, `NewDetKDecomp` can also compute an approximated form of FHDs which will be described in Section 4.3.5.

We designed a second Java library, called *hg-tools*, for analyzing SQL queries, XCSP instances, and collecting hypergraph statistics. In particular, this software implements all the functionalities presented in Section 4.1 and is thus able to transform simple and complex SQL queries as well as CSPs in the XCSP format into hypergraphs. These features have been implemented with the support of the open source libraries *JSqlParser* [139] for SQL processing and *JGraphT* [108] to deal with graph data structures. The software have also been extended to compute some hypergraph statistics we used for experiments. This code is available at `https://github.com/dmlongo/hgtools`.

All the experiments reported in this paper were performed on a cluster of 10 workstations each running Ubuntu 16.04. Every workstation has the same specification and is equipped with two Intel Xeon E5-2650 (v4) processors each having 12 cores and 256-GB main memory. Since all algorithms are single-threaded, we could run several experiments in parallel. For all upcoming runs of our algorithms we set a timeout of 3600s.

4.2.2 Hypergraph Benchmark

Our benchmark contains 3648 hypergraphs, which have been converted from CQs and CSPs collected from various sources. Out of these 3648 hypergraphs, 3142 hypergraphs have never been used in a hypertree width analysis before. The hypertree width of 424 CQs and of 82 CSPs has been analyzed in [76], [22], and/or [27, 28]. In particular, the hypergraphs of the SPARQL queries from [27, 28] have been kindly shared with us by the authors and their *hw* has already been analyzed in those papers. An overview of all instances of CQs and CSPs is given in Table 4.1. They have been collected from various publicly available benchmarks and repositories of CQs and CSPs. In the first column, the names of each collection of CQs and CSPs are given together with references where they were first published. In the second column we display the number of hypergraphs extracted from each collection. The *hw* of the CQs and CSPs in our benchmark will be discussed in detail in Section 4.3.1. To get a first feeling of the *hw* of the various sources, we mention the number of cyclic hypergraphs (i.e., those with $hw \geq 2$) in the last column. When gathering the CQs, we proceeded as follows: of the huge benchmark reported in [27], we have only included CQs, which were detected as having $hw \geq 2$ in [27]. Of the other huge repository reported in [28], we included the hypergraphs corresponding to the 273974 unique SPARQL queries with $hw \geq 2$. Even though the queries are unique, most of them share the same hypergraph structure. Thus, after removing duplicates on the hypergraph level, we ended up with 354 unique hypergraphs with $hw \geq 2$. Of the big repository reported in [90], we have included those CQs, which are not trivially acyclic (i.e., they have at least 3 atoms). Of all the small collections of queries, we have included all. It follows a detailed description of the different benchmarks.

Table 4.1: Overview of benchmark instances.

	Benchmark	No. instances	$hw \geq 2$
CQs	SPARQL [27]	70 (out of 26157880)	70
	WIKIDATA [28]	354 (out of 273947)	354
	LUBM [21, 80]	14	2
	iBENCH [21, 13]	40	0
	DOCTORS [21, 60]	14	0
	DEEP [21]	41	0
	JOB (IMDB) [101]	33	7
	TPC-H [20, 132]	29	1
	TPC-DS [131]	228	5
	SQLSHARE [90]	290 (out of 15170)	1
	RANDOM [118]	500	464
CSPs	APPLICATION [15]	1090	1090
	RANDOM [15]	863	863
	OTHER [76, 22]	82	82
	Total:	*3648*	*2939*

Our benchmark contains 1113 CQs from five main sources [20, 21, 27, 28, 90] and a set of 500 randomly generated queries using the query generator of [118]. In the sequel, we shall refer to the former queries as *CQ Application*, and to the latter as *CQ Random*. The CQs analyzed in [27] constitute a big repository of CQs – namely 26157880 CQs stemming from SPARQL queries. The queries come from real-users of SPARQL endpoints and their hypertree width was already determined in [27]. Almost all of these CQs were shown to be acyclic. Our analysis comprises 70 CQs from [27], which (apart from few exceptions) are essentially the ones in [27] with $hw \geq 2$. In particular, we have analyzed all 8 CQs with highest hw among the CQs analyzed in [27] (namely, $hw = 3$). Bonifati et al. carried on this line of work and examined a bigger repository of SPARQL queries coming from Wikidata in [28]. This repository of 208215209 SPARQL queries was originally released by Malyshev et al. with the study in [104]. Bonifati et al. kindly sent us the unique 273947 SPARQL queries with $hw \geq 2$ examined in [104]. We extracted 354 different hypergraphs and all of them have $hw = 2$.

The LUBM [80], iBENCH [13], DOCTORS [60], and DEEP scenarios have been recently used to evaluate the performance of chase-based systems [21]. Their queries were especially tailored towards the evaluation of query answering tasks of such systems. Note that the LUBM benchmark [80] is a widely used standard benchmark for the evaluation of Semantic Web repositories. Its queries are designed to measure the performance of those repositories over large datasets. Strictly speaking, the iBENCH is a tool for generating schemas, constraints, and mappings for data integration tasks. However, in [21], 40 queries were created for tests with the iBENCH. We therefore refer to these queries as iBench-CQs here. In summary, we have incorporated all queries that were either contained in the original benchmarks or created/adapted for the tests in [21].

The goal of the Join Order Benchmark (JOB) [101] was to evaluate the impact of a good join order on the performance of query evaluation in standard RDBMSs. Those queries were formulated over the real-world dataset Internet Movie Database (IMDB). All of the queries have between 3 and 16 joins. Clearly, as the goal was to measure the impact of a good join order, those 33 queries are of higher complexity, hence 7 out of the 33 queries have $hw \geq 2$.

The Transaction Processing Performance Council (TPC) is a well-known non-profit organization that develops benchmarks for the evaluation of DBMSs. Given their broad industry-wide relevance and since they reflect common workloads in decision support systems, we included the TPC-H [132] and the TPC-DS [131] benchmarks. The TPC-H queries analyzed in [50, 49] were downloaded from the GitHub repository originally provided by Michael Benedikt and Efthymia Tsamoura [20] for the work on [21]. Nevertheless, for this paper we downloaded the original dataset from [132] and extracted the queries according to the methodology introduced in Section 4.1. From the original set of 22 complex queries, we extracted 29 simple queries. The TPC-DS benchmark is more complex than TPC-H and it contains more queries. Indeed, from the original set of 113 complex queries, we extracted 228 simple queries.

From SQLShare [90], a multi-year SQL-as-a-service experiment with a large set of real-world queries, we extracted 15170 queries by considering all queries in the log files. These were divided into two sets: materialized views and usual queries which could possibly make use of the materialized views. Also, the whole dataset gathers data from different databases and the link between queries and databases is not explicitly defined. In order to execute the experiments, we had to clean the queries from trivial errors impeding the parsing, link the queries to the right database schema, incorporate the materialized views and resolve ambiguities in the query semantics. After removing queries with complex syntactical errors, we obtained 12483 queries. As a next step we used the algorithm from Section 4.1.2 to obtain a set of CQs and, after removing duplicates, we got a collection of 6086 simple SQL queries. From this set we eliminated 5796 queries with ≤ 2 atoms (whose acyclicity is immediate) and ended up with 290 queries.

The random queries were generated with a tool that stems from the work on query answering using views in [118]. The query generator allows three options: chain/star/random queries. Since the former two types are trivially acyclic, we only used the third option. Here it is possible to supply several parameters for the size of the generated queries. In terms of the resulting hypergraphs, one can thus fix the number of vertices, number of edges and arity. We have generated 500 CQs with 5–100 vertices, 3–50 edges and arities from 3 to 20. These values correspond to the values observed for the *CQ Application* hypergraphs. However, even though these size values have been chosen similarly, the structural properties of the hypergraphs in the two groups *CQ Application* and *CQ Random* differ significantly, as will become clear from our analysis in Section 4.3.1.

Table 4.2: Properties of all benchmark CQs.

CQ Application

i	Deg	IS	3-MIS	4-MIS	VC
0	0	74	394	647	72
1	74	721	673	456	484
2	320	286	45	10	557
3	253	17	1	0	0
4	181	6	0	0	0
5	73	9	0	0	0
>5	212	0	0	0	0

CQ Random

i	Deg	IS	3-MIS	4-MIS	VC
0	0	1	16	49	0
1	1	17	77	125	20
2	15	53	90	120	133
3	38	62	103	74	240
4	31	63	62	42	106
5	33	71	47	28	1
>5	382	233	105	62	0

Our benchmark currently contains 2035 hypergraphs from CSP instances, out of which 1953 instances were obtained from `xcsp.org` (see also [15]). These instances, in turn, were divided into CSPs from concrete applications, called *CSP Application* in the sequel (1090 instances), and randomly generated CSPs, called *CSP Random* below (863 instances). In addition, we have included 82 CSP instances, which were already used in previous *hw* experiments [57, 76]. These instances, which we refer to as *CSP Other*, are provided also provided at `https://www.dbai.tuwien.ac.at/proj/hypertree`.

Our HyperBench benchmark consists of the hypergraphs of these CQs and CSPs. In Figure 4.2, we show the number of vertices, the number of edges and the arity (i.e., the maximum size of the edges) as three important metrics of the size of each hypergraph. The smallest are those coming from *CQ Application* (most of them have up to 10 edges), while the hypergraphs coming from CSPs can be significantly larger (up to 2993 edges). Although some hypergraphs are very big, more than 50% of all hypergraphs have maximum arity less than 5. In Figure 4.2 we can easily compare the different types of hypergraphs, e.g. hypergraphs of arity greater than 20 only exist in the application classes; the *CSP Other* class contains the largest number of big hypergraphs.

The hypergraphs and the results of our analysis can be accessed through our web tool, available at `http://hyperbench.dbai.tuwien.ac.at`.

4.3 Comparison of GHD Decomposition Algorithms

We now present the empirical results obtained with the HyperBench benchmark. While getting an overview of the hypertree width of the various types of hypergraphs in our benchmark, we want to find out how realistic the restriction to low values for certain hypergraph invariants is. After this first analysis of structural properties, we compare the different *ghw* algorithms presented in Chapter 3. Finally, we propose and evaluate two algorithms for computing approximated FHDs.

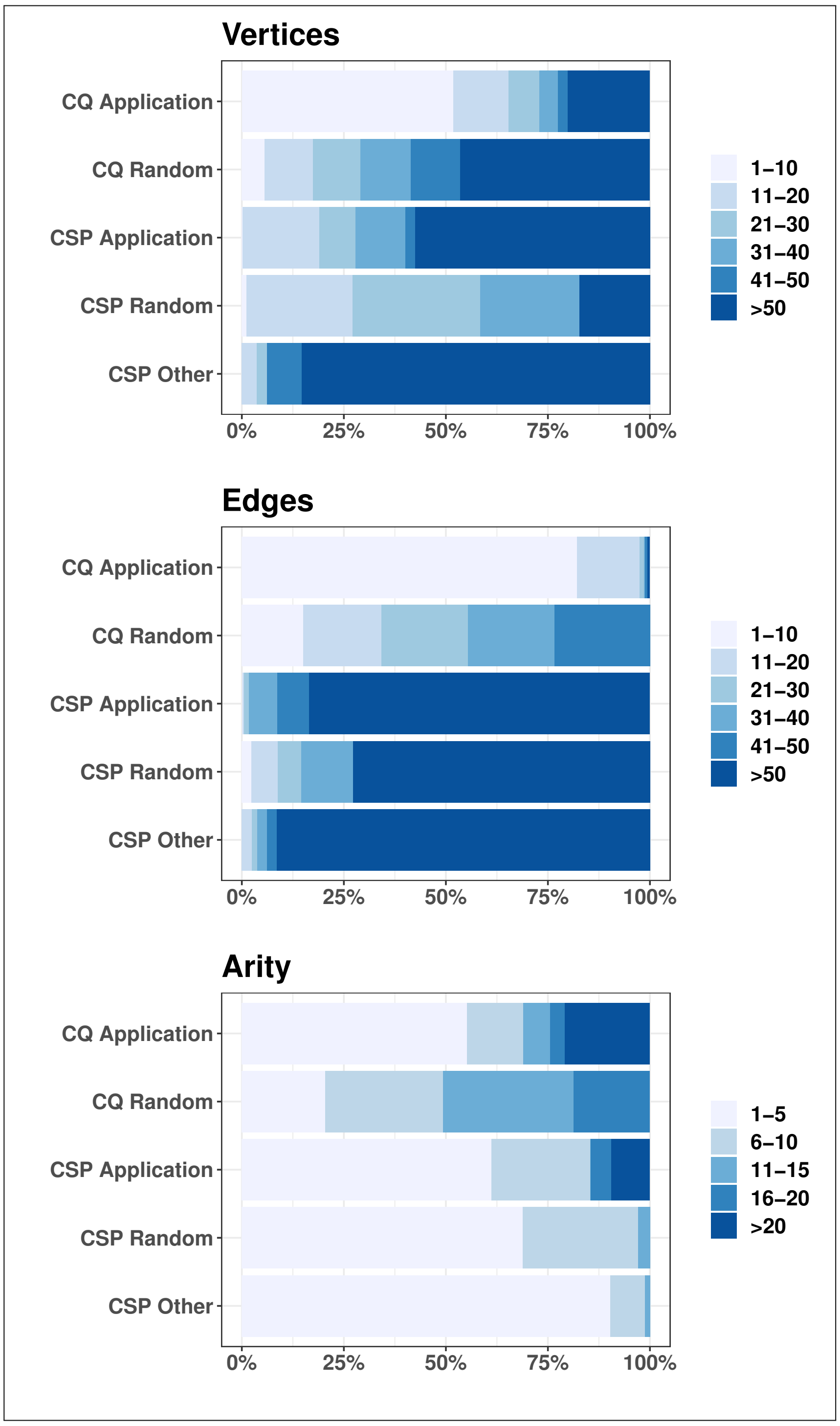


Figure 4.2: Hypergraph Sizes.

Table 4.3: Properties of all benchmark CSPs.

CSP Application

i	Deg	IS	3-MIS	4-MIS	VC
0	0	0	596	597	0
1	0	1030	459	486	0
2	596	59	34	7	1064
3	1	0	1	0	26
4	1	0	0	0	0
5	2	0	0	0	0
>5	490	1	0	0	0

CSP Random

i	Deg	IS	3-MIS	4-MIS	VC
0	0	0	0	0	0
1	0	200	200	238	0
2	0	224	312	407	220
3	0	76	147	95	515
4	12	181	161	97	57
5	8	99	14	1	71
>5	843	83	29	25	0

CSP Other

i	Deg	IS	3-MIS	4-MIS	VC
0	0	0	1	6	0
1	0	7	36	39	0
2	1	36	23	16	50
3	5	29	20	21	25
4	19	10	2	0	0
5	4	0	0	0	0
>5	53	0	0	0	0

4.3.1 Hypergraph Properties

In [53, 68], several invariants of hypergraphs were used to make the Check(ghw, k) and Check(fhw, k) problems tractable or, at least, easier to approximate. We thus investigate the following properties defined in Chapter 2:

- *Deg*: the degree of the underlying hypergraph.
- *IS*: the intersection size of two edges.
- *c-MIS*: the c-multi-intersection size for $c \in \{3, 4\}$.
- *VC*: the VC-dimension.

The results obtained from computing *Deg*, *IS*, 3-*MIS*, 4-*MIS*, and *VC* for the hypergraphs in the HyperBench benchmark are shown in Table 4.2 and Table 4.3.

These tables have to be read as follows. In the first column, we distinguish different values of the various hypergraph metrics. In the columns labeled "Deg", "IS", etc., we indicate for how many instances each metric has a certain value. For instance, the last row in the second column shows that only 212 non-random CQs have degree > 5. Actually, for most CQs, the degree is less than 10. Moreover, already with intersections of 3 edges, we get 3-MIS ≤ 2 for almost all non-random CQs. The VC-dimension is ≤ 2 for all hypergraphs.

Table 4.4: Exact number of CQ instances in the HW analysis of Figure 4.3.

CQ Application

k	Yes	No	T/O
1	673	440	0
2	432	8	0
3	8	0	0

CQ Random

k	Yes	No	T/O
1	36	464	0
2	68	396	0
3	70	326	0
4	59	167	100
5	54	55	158
10	206	0	7
15	7	0	0

For CSPs, all properties may have higher values. However, we note a significant difference between randomly generated CSPs and the rest. For hypergraphs in the groups *CSP Application* and *CSP Other*, 543 (46%) hypergraphs have a high degree (>5), but nearly all instances have IS or MIS of less than 3. And most instances have a VC-dimension of at most 2. In contrast, nearly all random instances have a significantly higher degree (843 out of 863 instances with a degree > 5). Nevertheless, many instances have small IS and MIS. For nearly all hypergraphs (838 out of 863) we have 4-multi-intersection size ≤ 4. For 7 instances the computation of the VC-dimension timed out. For all others, the VC-dimension is ≤ 5 for random CSPs. Clearly, as seen in Table 4.2 and Table 4.3, the random CQs resemble the random CSPs a lot more than the CQ and CSP Application instances. For example, random CQs have similar to random CSPs high degree (382, corresponding to 76%, with degree > 5), higher IS and MIS. Nevertheless, similarly to random CSPs, the values for IS and MIS are still small for many random CQ instances.

To conclude, for the proposed properties, in particular IS/MIS and VC-dimension, most of the hypergraphs in our benchmark indeed have low values.

4.3.2 Hypertree Width

We have systematically applied the *hw*-computation from [76] to all hypergraphs in the benchmark. The results are summarized in Figure 4.3. The exact number of instances for each stage of the experiments is reported in Table 4.4 for CQs and Table 4.5 for CSPs. The acronym "T/O" stands for "Timeout". In our experiments, we proceeded as follows. We used the same classification of instances we used in the previous experiments, i.e., we distinguished the following classes: *CQ Application*, *CQ Random*, *CSP Application*, *CSP Random*, and *CSP Other*. For every hypergraph H, we first tried to solve the Check(hw, k) problem for $k = 1$. In case of *CQ Application*, we thus got 673 yes-answers and 440 no-answers. The number in each bar indicates the average runtime to find these yes- and no-instances, respectively. Here, the average runtime was "0" (i.e., less than 1 second). For *CQ Random* we got 36 yes- and 464 no-instances with an average runtime below 1 second. For all CSP-instances, we only got no-answers.

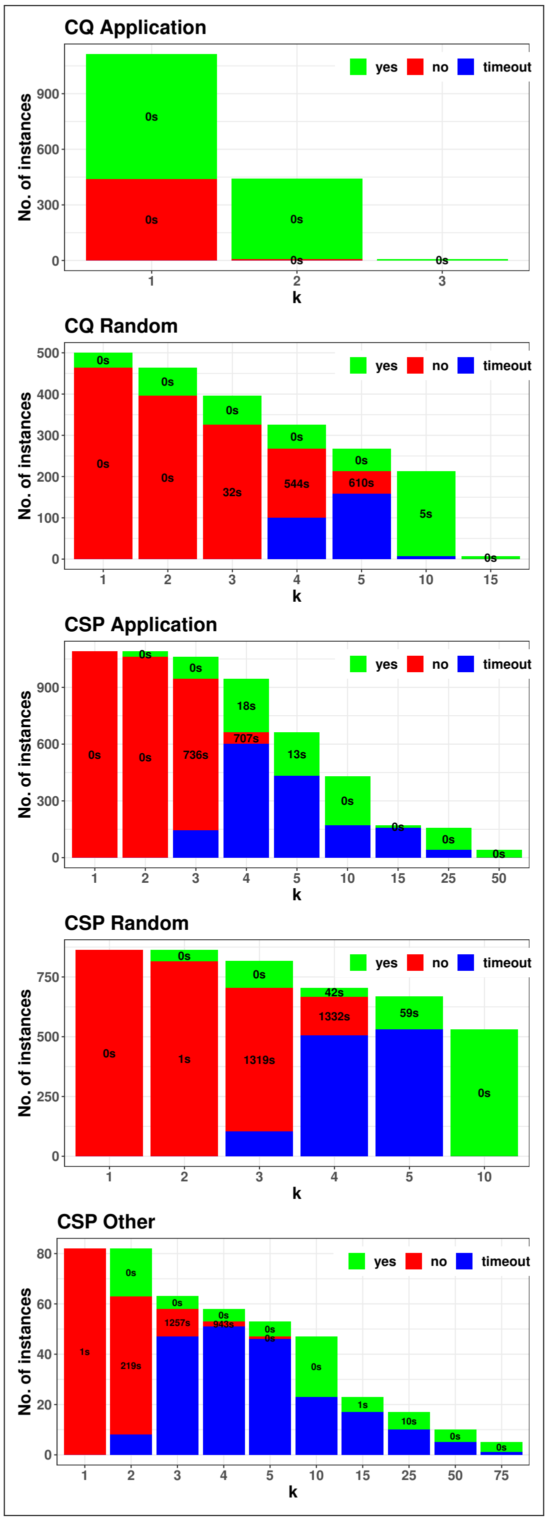


Figure 4.3: HW analysis (labels are avg. runtimes in s).

Table 4.5: Exact number of CSP instances in the HW analysis of Figure 4.3.

CSP Application

k	Yes	No	T/O
1	0	1090	0
2	29	1061	0
3	116	802	143
4	283	62	600
5	231	0	431
10	261	0	170
15	12	0	158
25	118	0	40
50	40	0	0

CSP Random

k	Yes	No	T/O
1	0	863	0
2	47	816	0
3	111	602	103
4	39	160	506
5	136	0	530
10	530	0	0

CSP Other

k	Yes	No	T/O
1	0	82	0
2	19	55	8
3	5	11	47
4	5	2	51
5	6	1	46
10	24	0	23
15	6	0	17
25	7	0	10
50	5	0	5
75	4	0	1

In the second round, we tried to solve the Check(hw, k) problem for $k = 2$, for all hypergraphs that yielded a no-answer for $k = 1$. Here the picture is a bit more diverse: 432 of the remaining 440 CQs from *CQ Application* yielded a yes-answer in less than 1 second. For the hypergraphs stemming from *CQ Random*, only 68 instances yielded a yes-answer (in less than 1 second on average), while 396 instances yielded a no-answer in less than 7 seconds on average. The hypergraphs relative to CSPs tell a different story. The classes *CSP Application*, *CSP Random* and *CSP Other* have 29, 47 and 19 yes-instances, respectively. Only 8 instances from *CSP Other* gave rise to a timeout (i.e., the program did not terminate within 3,600 seconds), while all the other instances gave a no-answer within the timeout. Interestingly, the hw-algorithm gave a no-answer for 1877 instances of *CSP Application* and *CSP Random* in less than 1 second, while it took the algorithm 219 seconds on average to answer "no" for 55 instances of *CSP Other*. This shows that the class *CSP Other* contains instances which are difficult to decompose, in fact much more than the hypergraphs in the other classes.

This procedure was iterated by incrementing k and running the hw-computation for all those instances that either yielded a no-answer or a timeout in the previous round. For instance, for queries from *CQ Application*, one further round was needed after the second round. In other words, we confirm the observation of low hw, which was already made for CQs of arity ≤ 3 in [27, 28, 117]. For the hypergraphs stemming from *CQ Random* (resp. CSPs), 396 (resp. 1940) instances were left in the third round, of which 70 (resp. 232) yielded a yes-answer in less than 1 second on average, 326 (resp. 1415) instances yielded a no-answer in 32 (resp. 988) seconds on average and no (resp. 293) instances yielded a timeout. Note that, as we increased k, the average runtime and the percentage of timeouts first increased up to a certain point and then they decreased. This is due to the fact that, as we increase k, the number of combinations of edges to be considered in each edge cover λ (i.e., the function λ_u at each node u of the decomposition) increases. In principle, we have to test $\mathcal{O}(n^k)$ combinations, where n is the number of edges. However,

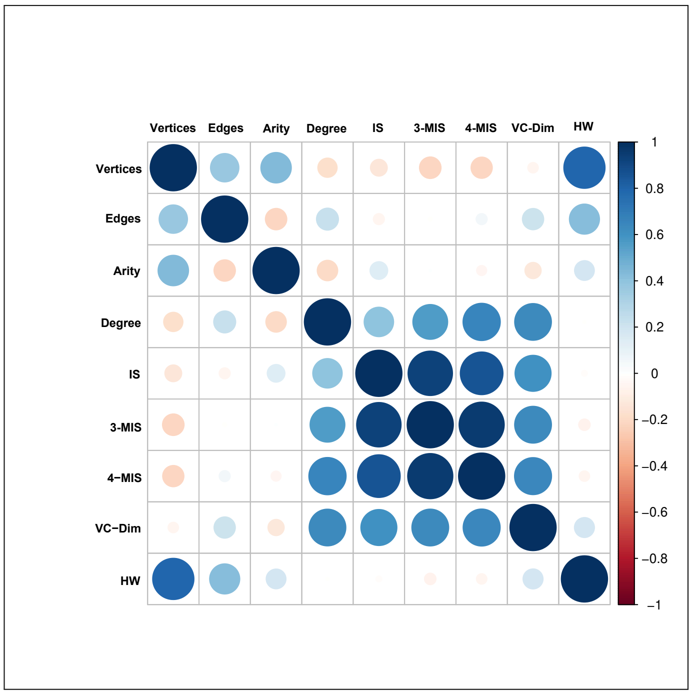


Figure 4.4: Correlation analysis of hypergraph properties.

if k increases beyond a certain point, then it gets easier to "guess" an edge cover λ since an increasing portion of the $\mathcal{O}(n^k)$ possible combinations leads to an HD of desired width.

Looking at the results of these experiments, we conclude that for a big number of instances, the hypertree width is small enough to allow for efficient evaluation of CQs or CSPs: all instances of non-random CQs have $hw \leq 3$ no matter whether their arity is bounded by 3 (as in case of SPARQL queries) or not; and a large portion (at least 1027, i.e., ca. 50%) of all 2035 CSP instances have $hw \leq 5$. In total, including random CQs, 2427 (66.5%) out of 3648 instances have $hw \leq 5$. And, out of these, we could determine the exact hypertree width for 2356 instances; the others may even have lower hw.

4.3.3 Correlation Analysis

Finally, we analyzed the pairwise correlation between all properties. Of course, the different intersection sizes (IS, 3-MIS, 4-MIS) are highly correlated. Other than that, we observe quite a strong correlation of the arity with number of vertices and hypertree width. Moreover, there is a significant correlation between number of vertices and arity and between number of vertices and hypertree width. Clearly, the correlation between arity and hypertree width is mainly due to the CSP instances and the random CQs since, for non-random CQs, the hw never increases beyond 3, independently of the arity.

A graphical presentation of all pairwise correlations is given in Figure 4.4. Large dark circles indicate a high correlation, while small light circles stand for low correlation. Blue

Table 4.6: GHW algorithms with avg. runtimes in s.

hw → *ghw*	*Total*	`GlobalBIP`		`LocalBIP`		`BalSep`	
		yes (s)	*no (s)*	*yes (s)*	*no (s)*	*yes (s)*	*no (s)*
3 → 2	310	-	128 (537)	-	195 (162)	-	307 (12)
4 → 3	386	-	137 (2809)	-	54 (2606)	-	249 (54)
5 → 4	427	-	-	-	-	-	148 (13)
6 → 5	459	13 (162)	-	13 (60)	-	-	180 (288)

circles indicate a positive correlation while red circles stand for a negative correlation. It has been argued in [53] that Deg, IS, 3-MIS, 4-MIS and VC are non-trivial restrictions to achieve tractability. It is interesting to note that, according to Figure 4.4, these properties have almost no impact on the hypertree width of our hypergraphs. This underlines the usefulness of these restrictions in the sense that (a) they make the GHD computation and FHD approximation easier [53], but (b) low values of degree, (multi-)intersection-size, or VC-dimension do not pre-determine low values of the widths.

4.3.4 Comparison of *ghw* Algorithms

Here we report on empirical results for the three *ghw*-algorithms described in Chapter 3. We have run the programs on each hypergraph from the HyperBench up to hypertree width 6, trying to get a smaller *ghw* than *hw*. We have thus run the *ghw*-algorithms with the following parameters: for all hypergraphs H with $hw(H) = k$ (or $hw \leq k$ and, due to timeouts, we do not know if $hw \leq k-1$ holds), where $k \in \{3, 4, 5, 6\}$, try to solve the $\textsc{Check}(ghw, k-1)$ problem. In other words, we just tried to improve the width by 1. Clearly, for $hw(H) \in \{1, 2\}$, no improvement is possible since, in this case, $hw(H) = ghw(H)$ holds.

In Table 4.6, for each algorithm, we report on the number of "successful" attempts to solve the $\textsc{Check}(ghw, k-1)$ problem for hypergraphs with $hw = k$. Here "successful" means that the program terminated within 1 hour. For instance, for the 310 hypergraphs with $hw = 3$ in the HyperBench, `GlobalBIP` terminated in 128 cases (i.e., 41%) when trying to solve $\textsc{Check}(ghw, 2)$. The average runtime of these "successful" runs was 537 seconds. For the 386 hypergraphs with $hw = 4$, `GlobalBIP` terminated in 137 cases (i.e., 35%) with average runtime 2809 when trying to solve the $\textsc{Check}(ghw, 3)$ problem. For the 886 hypergraphs with $hw \in \{5, 6\}$, `GlobalBIP` only terminated in 13 cases (i.e., 1.4%). Overall, it turns out that the set $f(H, k)$ may be very big (even though it is polynomial if k and i are constants). Hence, H' can become considerably bigger than H. This explains the frequent timeouts in the `GlobalBIP` column in Table 4.6.

The results obtained with `LocalBIP` are shown in the corresponding column. Interestingly, for the hypergraphs with $hw = 3$, the "local" computation performed significantly better (namely 63% solved with average runtime 162 seconds rather than 41% with average runtime 537 seconds). In contrast, for the hypergraphs with $hw = 4$, the "global"

Table 4.7: GHW of instances with average runtime in s.

$hw \to ghw$	Yes	No	T/O
$3 \to 2$	0	309 (10)	1
$4 \to 3$	0	262 (57)	124
$5 \to 4$	0	148 (13)	279
$6 \to 5$	18 (129)	180 (288)	261

computation was significantly more successful. For $hw \in \{5, 6\}$, the "global" and "local" computations were equally bad. A possible explanation for the reverse behavior of "global" and "local" computation in case of $hw = 3$ as opposed to $hw = 4$ is that the restriction of the "global" set $f(H, k)$ of subedges to the "local" set $f_u(H, k)$ at each node u seems to be quite effective for the hypergraphs with $hw = 3$. In contrast, the additional cost of having to compute $f_u(H, k)$ at each node u becomes counter-productive, when the set of subedges thus eliminated is not significant. It is interesting to note that the sets of solved instances of the global computation and the local computation are incomparable, i.e., in some cases one method is better, while in other cases the other method is better.

If we look at the number of solved instances in Table 4.6, we see that the recursive algorithm via balanced separators (reported in the last column labeled `BalSep`) has the least number of timeouts due to the fast identification of negative instances (i.e., those with no-answer), where it often detects quite fast that a given hypergraph does not have a balanced separator of desired width. As k increases, the performance of the balanced separators approach deteriorates. This is due to k in the exponent of the running time of our algorithm, i.e. we need to check for each of the possible $\mathcal{O}(n^{k+1})$ combinations of $\leq k$ edges if it constitutes a balanced separator. Note that the balanced separators approach only terminated in case of no-answers.

We report in Table 4.7 whether $ghw \leq k - 1$ could be verified, for all hypergraphs with $hw \leq k$ and $k \in \{3, 4, 5, 6\}$. We thus run the three algorithms (`GlobalBIP`, `LocalBIP` and `BalSep`) in parallel and stopped the computation as soon as one terminated (with answer "yes" or "no"). The number in parentheses refers to the average runtime needed by the fastest of the three algorithms. A timeout occurs if no algorithm terminates within 3600 seconds. It is interesting to note that in the vast majority of cases, no improvement of the width is possible when we switch from hw to ghw: in 97% of the solved cases with $hw \leq 6$, which form 65% of all instances, hw and ghw have identical values. Actually, we think that the high percentage of the *solved cases* gives a more realistic picture than the percentage of *all cases* for the following reason: our algorithms (in particular, the "global" and "local" computations) need particularly long time for negative instances. This is due to the fact that in a negative case, "all" possible choices of edge covers λ for a node u in the GHD have to be tested before we can be sure that no GHD of H (or, equivalently, no HD of H') of desired width exists. Hence, it seems plausible that the timeouts are mainly due to negative instances. This also explains why our new `BalSep` algorithm, which is particularly well suited for negative instances, has the least number of timeouts.

A closer comparison of Table 4.6 and Table 4.7 makes clear that `BalSep` is superior to `GlobalBIP` and `LocalBIP` in solving negative instances. Indeed, the combined approach summarized in Table 4.7 relies almost completely on the runs of `BalSep`. The algorithms `GlobalBIP` and `LocalBIP` managed to solve only 15 out of 571 negative instances for $k \in \{3, 4\}$, while they could not give any negative answer for $k \in \{5, 6\}$. On the other hand, `GlobalBIP` and `LocalBIP` solved 18 positive instances, while `BalSep` did not terminate at all. Nevertheless, this does not significantly diminish the strength of `BalSep` as a powerful tool for negative instances.

We conclude with a final observation: in Figure 4.3, we had many cases, for which only some upper bound k on the hw could be determined, namely those cases, where the attempt to solve $\text{Check}(hw, k)$ yielded a yes-answer and the attempt to solve $\text{Check}(hw, k-1)$ gave a timeout. In several such cases, we could get (with the balanced separator approach) a no-answer for the $\text{Check}(ghw, k-1)$ problem, which implicitly gives a no-answer for the problem $\text{Check}(hw, k-1)$. In this way, our new ghw-algorithm is also profitable for the hw-computation: for 827 instances with $hw \leq 6$, we were not able to determine the exact hypertree width. Using our new ghw-algorithm, we closed this gap for 297 instances; for these instances $hw = ghw$ holds.

To sum up, we now have a total of 2356 (64.5%) instances for which we determined the exact hw and a total of 1984 instances (54.4%) for which we determined the exact ghw. Out of these, 1968 instances had identical values for hw and ghw. In 16 cases, we found an improvement of the width by 1 when moving from hw to ghw, namely from $hw = 6$ to $ghw = 5$. In 2 further cases, we could show $hw \leq 6$ and $ghw \leq 5$, but the attempt to check $hw = 5$ or $ghw = 4$ led to a timeout. Hence, $hw = ghw$ in 54.4% of the cases if we consider all instances and in 68.2% of the cases (1968 of 2886) with small width ($hw \leq 6$). However, if we consider the fully solved cases (i.e., where we have the precise value of hw and ghw), then hw and ghw coincide in 99.2% of the cases (1968 of 1984).

4.3.5 Fractionally Improved Decompositions

Computing FHDs is very expensive even in tractable cases, as the result in [53] involves a double exponential "constant". Here we propose two algorithms for computing an FHD of a hypergraph when we already have a GHD: `ImproveHD` and `FracImproveHD`. They differ in the compromise between computational cost and quality of the approximation.

The first algorithm we present is based on a simple observation: given a GHD, we could substitute its integral edge covers with fractional edge covers and obtain an FHD. Formally, let $\mathcal{D} = \langle T, (B_u)_{u \in T}, (\lambda_u)_{u \in T} \rangle$ be either a GHD or an HD. Our algorithm `ImproveHD` computes an FHD $\mathcal{D}' = \langle T', (B'_u)_{u \in T'}, (\gamma_u)_{u \in T'} \rangle$ where:

- The tree T' is the same as T.
- For each node $u \in T'$, the bag $B'_u = B_u$.
- For each node $u \in T'$, γ_u is a minimum-weight fractional edge cover of B'_u.

Table 4.8: Instances solved with `ImproveHD`.

hw	≥ 1	$[0.5, 1)$	$[0.1, 0.5)$	No	T/O
2	0	136	40	419	0
3	12	104	25	169	0
4	9	55	11	311	0
5	20	14	11	382	0
6	12	60	80	307	0

Table 4.9: Instances solved with `FracImproveHD`.

hw	≥ 1	$[0.5, 1)$	$[0.1, 0.5)$	No	T/O
2	0	194	46	353	2
3	14	116	21	135	24
4	11	81	2	8	284
5	18	126	59	2	222
6	28	149	95	4	183

To obtain the FHD $\mathcal{D}'$, we iterate over the nodes of $\mathcal{D}$ and, for each B_u, we compute a minimum-weight fractional edge cover of B_u. Since computing such a fractional edge cover is polynomial and we assume to have already computed an HD to start with, the whole algorithm is efficient. Nevertheless, it is clear that there is a strong dependence on the starting HD. This is unsatisfactory and so we devised a more sophisticated algorithm.

The algorithm we describe here gets rid of the dependence on a particular HD and computes a fractionally improved (G)HD with a fixed improvement threshold. We call this algorithm `FracImproveHD`. It searches for an FHD $\mathcal{D}'$ with $\mathcal{D}' = \texttt{ImproveHD}(\mathcal{D})$ for some HD $\mathcal{D}$ of H with $width(\mathcal{D}) \leq k$ and $width(\mathcal{D}') \leq k'$. Here, k is an upper bound on the hw and k' the desired fractionally improved hw. In other words, this algorithm searches for the best fractionally improved HD over all HDs of width $\leq k$. Hence, the result is independent of any concrete HD.

The algorithm `FracImproveHD` is built on top of the GHD construction described in Chapter 2. Recall that, given a hypergraph H, this algorithm maintains a set of edges $C \subseteq E(H)$ which represents the current component to decompose. While searching for a separator λ_u of C, we do not only want that $|\lambda_u| \leq k$, but we also require that, among all possible choices of λ_u, we choose one such that $weight(\gamma_u) \leq k'$, where γ_u is a fractional edge cover of $B(\lambda_u)$. This guarantees that the output is the desired FHD.

The experimental results with these algorithms for computing fractionally improved HDs are summarized in Table 4.8 and Table 4.9. We have applied these algorithms to all hypergraphs for which $hw \leq k$ with $k \in \{2, 3, 4, 5, 6\}$ is known from Figure 4.3. The various columns of these tables are as follows: the first column (labeled hw) refers to the (upper bound on the) hw according to Figure 4.3. The next 3 columns, labeled ≥ 1,

$[0.5, 1)$, and $[0.1, 0.5)$ tell us, by how much the width can be improved (if at all) if we compute an FHD by one of the two algorithms. We thus distinguish the 3 cases if, for a hypergraph of $hw \leq k$, we manage to construct an FHD of width $k - c$ for $c \geq 1$, $c \in [0.5, 1)$, or $c \in [0.1, 0.5)$. The column with label "No" refers to the cases where no improvement at all or at least no improvement by $c \geq 0.1$ was possible. The last column counts the number of timeouts.

For instance, in the first row of Table 4.8, we see that (with the `ImproveHD` algorithm and starting from the HD obtained by the *hw*-computation of Figure 4.3) out of 595 hypergraphs with $hw = 2$, no improvement was possible in 419 cases. In the remaining 176 cases, an improvement to a width of at most $2 - 0.5$ was possible in 40 cases and an improvement to $k - c$ with $c \in [0.1, 0.5)$ was possible in 136 cases. For the hypergraphs with $hw = 3$ in Figure 4.3, almost half of the hypergraphs (141 out of 310) allowed at least some improvement, in particular, 104 by $c \in [0.5, 1)$ and 12 even by at least 1. The improvements achieved for the hypergraphs with $hw \leq 4$ and $hw \leq 5$ are less significant.

The results obtained with our `FracImproveHD` implementation are displayed in Table 4.9. We see that the number of hypergraphs which allow for a fractional improvement of the width by at least 0.5 or even by 1 is often bigger than with `ImproveHD` – in particular in the cases where $k' \leq k$ with $k \in \{4, 5\}$ holds. In the other cases, the results obtained with the naive `ImproveHD` algorithm are not much worse than with the more sophisticated `FracImproveHD` algorithm.

4.4 Summary

In this chapter, we studied the problem of benchmarking decomposition algorithms, emphasizing the nature and structure of the instances to decompose. We based our study on *HyperBench*, a preexisting collection of hypergraphs originating from real-world CQs and CSPs. We recognized that HyperBench lacked CQ hypergraphs, as most instances came from CSPs. Moreover, the CQs consisted of simple SQL queries to the detriment of more complex SQL queries and queries written in different languages, such as SPARQL. Therefore, we collected a more varied sample of complex SQL and SPARQL queries to balance this dataset.

The extraction of hypergraphs from complex SQL queries proved to be a complicated task. Since using existing methods for non-basic SQL queries was impossible, we developed a new methodology for transforming complex SQL queries into hypergraphs. At first, we distinguished between simple and complex SQL queries. The latter are those using constructs of the SQL language hiding the essential conjunctive structure. Nevertheless, it is not always possible to rewrite a query in conjunctive form, but it is often meaningful to extract "maximal conjunctive components" and analyze them separately. We proposed and implemented an algorithm for this case. On the front of SPARQL queries, we integrated an existing dataset of hypergraphs stemming from Wikidata queries. In the end, we rebalanced the partition between CQs and CSPs in the new HyperBench.

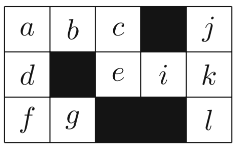
It followed a thorough analysis of the structural properties of the new HyperBench, which revealed that these hypergraphs typically have low width and (multi-)intersection sizes. Hence, real-world problems tend to have suitable characteristics for being solved via decompositions. Moreover, we performed an extensive benchmarking of `BalSep` and other GHD algorithms presented in Chapter 3. These experiments provide empirical evidence that GHD algorithms are highly effective in decomposing real-world hypergraphs. In particular, `BalSep` performed substantially better than other algorithms. To conclude, decomposition algorithms were able to successfully decompose real-world hypergraphs independently from the structure of CQs and CSPs and their nature.

CHAPTER 5

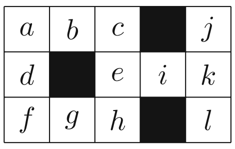
Updating GHDs upon Instances' Modifications

In Chapter 3 and Chapter 4, we proved that the `BalSep` algorithm is theoretically sound and can efficiently compute GHDs for a rich and varied set of hypergraphs. This fact makes `BalSep` an effective tool for decomposing hypergraphs in a typical static scenario where the input is given at the beginning of the computation and does not change over time. Indeed, the same decomposition is reusable several times to answer the same query over different databases or over the same database updated over time. Likewise, we can reuse an already computed decomposition for a CSP where the constraint relations, but not the constraint scopes, have been modified. This feature of GHDs makes it worth investing additional computational resources in finding a low-width GHD to reuse for future occasions.

Nevertheless, there are problems where the structure of an instance changes during the computation, then a solving approach based on structural decomposition methods would fail. For instance, *incremental constraint satisfaction* is a problem where the constraint solver needs to handle mutable sets of variables [127] or constraints [54]. If we were to solve this problem through decompositions, we would need to compute a new GHD of the instance every time it is modified, even slightly. While such a strategy could still be feasible for small hypergraphs with low width, where `BalSep` performs efficiently, we observed in Chapter 4 that especially the hypergraphs stemming from CSPs tend to have a higher number of edges and higher width in comparison to hypergraphs deriving from query answering problems. In this case, using the decomposition algorithms available in the literature like [76, 52, 75, 126] to compute low-width decompositions in response to every modification of the input would be detrimental for the performance. Indeed, these algorithms were not designed for a dynamic scenario where the hypergraph is repeatedly modified. In fact, to the best of our knowledge, this problem has not yet received any attention from the research community.

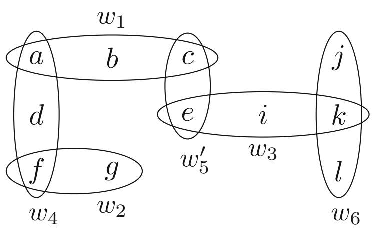


(a) Puzzle P.

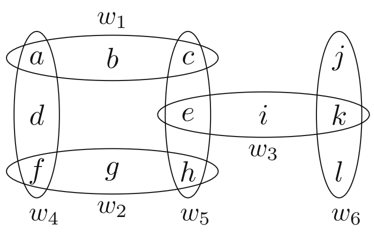


(b) Puzzle P'.

Figure 5.1: Two similar crossword puzzles P and P'. Given a set of words W, we want to fill every contiguous horizontal or vertical line of white cells with words from W. If two lines intersect, the words assigned to these lines must intersect in the right positions.



(a) Hypergraph H_P of P.



(b) Hypergraph $H_{P'}$ of P'.

Figure 5.2: The hypergraphs corresponding to the two puzzles P, P' of Figure 5.1.

To exemplify the issues encountered in a dynamic decomposition scenario, we introduce the following example, which we will use for the rest of the chapter.

Example 5.1. *Recall that a crossword puzzle, as introduced in Example 2.2, has to be filled with words belonging to a specific set. Each word must be inserted either horizontally or vertically so that the length of the word corresponds precisely to the number of contiguous white cells in a horizontal or vertical line. Also, intersecting lines must be filled with words having the same letter in the intersecting position. We model such a problem as a CSP where each cell is a variable and a constraint is defined over each contiguous line of white cells.*

Now consider the two crossword puzzles in Figure 5.1 and the respective hypergraphs in Figure 5.2. Suppose we first want to solve the puzzle P and then the slightly modified puzzle P' with the help of GHDs. Can we reuse the already-computed GHD of P and adjust only the parts affected by the modification to solve P'?

Although the puzzles of Example 5.1 closely resemble each other, using a GHD of P to solve P' is impossible. Thus, even if the hypergraphs of P and P' are almost identical, we must compute a new GHD to solve P' from scratch. On the other hand, obtaining a GHD of P' by slightly modifying the GHD of P already in our possession should be intuitively possible, if we have information about the difference between P and P'.

In this chapter, we discuss the problem of updating a GHD consequently to changes to the original hypergraph for which it was computed. Since the CSP area offers relevant applications for this problem, we develop our theory in a CSP context yet keep in mind that we can translate all of our findings into a database setting. We first develop a framework for constraint modifications and describe how these affect the underlying hypergraph. Here the focus is on capturing *elementary modifications* of CSPs, i.e., changes such as binding a variable to a constant, introducing a new constraint, and enforcing equality between a set of variables. We also define their dual operations. We then define the SearchUpdateGHD problem, i.e., the problem of updating a GHD when the original hypergraph changes w.r.t. the set of elementary modifications previously defined. Moreover, we study the complexity of the SearchUpdateGHD problem. The problem turns out to be not solvable in polynomial time for most modifications. Therefore, theoretically, updating a GHD is just as difficult as computing a new GHD for the modified hypergraph from scratch. Despite the hardness of the problem, we develop a practical method to deal with instance updates. Thus, we formulate a framework for efficient GHD updates that work with any modification. We back up this framework with an implementation strategy for adapting any existing top-down algorithm for computing GHDs to encompass also the update case. Finally, we extensively compare our method to classical algorithms. Given a modification, we use our algorithm to update an existing decomposition and compare it to a classical algorithm computing a new GHD for the modified instance. The results of these experiments show that our algorithm outperforms existing methods. In particular, we achieve mean speed-ups between a factor of 6 and 50 over the reference algorithm.

The rest of the chapter proceeds as follows. In Section 5.1, we define and study the SearchUpdateGHD problem. Here we first delineate a set of elementary modifications for CSPs and point out how the underlying hypergraph structure reflects these modifications. Then we investigate the complexity of the SearchUpdateGHD under the modifications previously defined. Next, we introduce a theoretical framework for handling instances updates and accordingly recompute only parts of a GHD affected by the update in Section 5.2. This framework of δ-mutable subtrees serves as a basis for defining a practical implementation strategy, which we discuss in Section 5.3. Here we focus on a high-level description of how to extend a top-down algorithm for computing GHDs to the update case. In Section 5.4, we implement this strategy for a reference algorithm and compare its performances against the original version with no support for updates. Here we show the results of our experiments. Finally, we summarize our findings in Section 5.6.

The contents of this chapter are based on the article [65], written by the author of this thesis in collaboration with Georg Gottlob, Matthias Lanzinger, and Cem Okulmus.

5.1 The GHD Update Problem

In this section, we propose relevant classes of elementary modifications of CSPs and describe their effect on both the CSP and the hypergraph. Then, we formally define the GHD update problem and settle its complexity.

5.1.1 Elementary Constraint Modifications

The modification of a CSP affects its underlying hypergraph and consequently its GHDs. We define a framework that allows us to update the GHD of a CSP in the face of constraint modifications. First of all, we define modifications.

Definition 5.1. *Given a hypergraph H, a modification is a function mapping hypergraphs to hypergraphs.*

We identify three fundamental hypergraph objects for the computation of a GHD: vertices, edges, and intersections between edges. As modifying a CSP typically implies the modification of these objects, we define six classes of modifications that reflect elementary changes on a hypergraph. In this sense, the proposed classes of modifications are natural, even though not necessarily minimal. We show the effect of these modifications on the hypergraph and intuitively explain their correspondence with the related CSP. On the other hand, we ignore all those CSP modifications that do not change the hypergraph. For instance, changing constraint relations does not affect the CSP structure, therefore the same GHD can be reused to solve the modified CSP again.

Hypergraph vertices can be either added or removed. We capture these possibilities in the following two classes for vertex modifications.

Definition 5.2. *AddVar is the class of modifications δ s.t. for every hypergraph H, given a new vertex $w \notin V(H)$:*

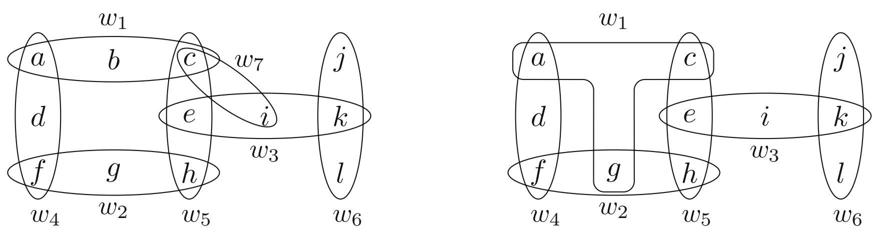
- $V(\delta(H)) \coloneqq V(H) \cup \{w\}$,
- $E(\delta(H)) \coloneqq E \cup \{e' \cup \{w\} \mid e' \in E'\}$, *where* (E, E') *is a partition of* $E(H)$ *with* $E' \neq \emptyset$.

DelVar is the class of modifications δ s.t. for every hypergraph H, given an existing vertex $v \in V(H)$, $V(\delta(H)) \coloneqq V(H) \setminus \{v\}$ and $E(\delta(H)) \coloneqq \{e \setminus \{v\} \mid e \in E(H)\}$.

On the CSP level, DelVar functions bind a CSP variable to a constant value and could possibly simplify its hypergraph. On the other hand, AddVar functions remove such binding, i.e., replace a constant with a variable. Alternatively, AddVar modifications can be seen as simply adding a new variable to an arbitrary number of constraints.

Analogously to the case of vertices, two classes for edge insertion and removal are now introduced.

Definition 5.3. *AddConstr is the class of modifications δ s.t. for every hypergraph H, $\delta(H) \coloneqq H \cup \{f\}$, where $f \notin E(H)$ is a new edge. DelConstr is the class of modifications δ s.t. for every hypergraph H, $\delta(H) \coloneqq H \setminus \{e\}$, where $e \in E(H)$.*



(a) H_{P_2} obtained using $\delta \in$ AddConstr. (b) H_{P_3} obtained using $\delta \in$ AddEq.

Figure 5.3: Hypergraphs obtained by applying to $H_{P'}$ the modifications described in Example 5.2.

AddConstr and DelConstr modifications correspond to alterations of the set of constraints C_t of a CSP. In particular, $\delta \in$ AddConstr introduces a new constraint in C_t, while $\delta \in$ DelConstr removes a constraint from C_t.

Finally, we present classes to modify intersections between edges. Let H be a hypergraph. Given $U \subseteq V(H)$, we denote with $E_U = \{e \in E(H) \mid e \cap U \neq \emptyset\}$ the edges incident on U.

Definition 5.4. *AddEq is the class of modifications δ s.t. for every hypergraph H, some vertices $U \subseteq V(H)$ are merged into $w \in V(\delta(H))$ and the edges in E_U are incident on w. DelEq is the class of modifications δ s.t. for every hypergraph H, a vertex $w \in V(H)$ is split into a set $U \subseteq V(\delta(H))$ and the edges in $E_{\{w\}}$ are arbitrarily distributed on U.*

Intuitively, an AddEq modification introduces an equality constraint between some variables of the CSP. In other words, a new *AllEqual* constraint is defined over a set of variables of the CSP. On the other hand, DelEq modifications remove this kind of constraint and thus all equalities between a specific set of variables.

Example 5.2. *The hypergraph $H_{P'}$ of Figure 5.2b is obtained by applying a modification $\delta \in$ AddVar to H_P in Figure 5.2a. In particular, δ adds a new vertex h in the edges $\{f, g\}$ and $\{c, e\}$ as well as in $V(H_P)$. Note that H_P can be obtained from $H_{P'}$ via a modification $\delta \in$ DelVar removing h from $V(H_{P'})$.*

Figure 5.3 shows two additional modifications of $H_{P'}$. The hypergraph H_{P_2} of Figure 5.3a is the result of a modification $\delta \in$ AddConstr introducing a new edge $\{c, i\}$, while H_{P_3} of Figure 5.3b shows the effect on $H_{P'}$ of a $\delta \in$ AddEq adding an AllEqual *constraint between the variables b, g (b is merged into g). Finally, $H_{P'}$ can be obtained through an appropriate inverse modification δ to H_{P_2} and H_{P_3} with $\delta \in$ DelConstr and $\delta \in$ DelEq, respectively.*

Note that the set of all considered elementary modifications is *complete*, i.e., given any two hypergraphs H, H', there always exists a sequence of elementary modifications $\delta_1, \ldots, \delta_\ell$ such that $H' = \delta_\ell(\cdots(\delta_1(H)))$.

5.1.2 The Complexity of Updating GHDs

Recall that checking $ghw(H) \leq k$, and therefore computing a width k GHD of H, is $\mathcal{NP}$-hard even when $k > 1$ is constant [53, 73, 69]. In the context of modifications this naturally presents the question of the complexity of the following task: given a hypergraph H together with a minimal width GHD, as well as a modification δ, find a GHD for $\delta(H)$ with the same width if one exists, or correctly identify that the width increased. Intuitively, the knowledge of a witness for H could make the problem easier, in particular if δ is a simple modification. Formally, we extend the standard problem of checking $ghw(H) \leq k$ for constant k (see e.g., [53]) by simply adding a modification (from some class of modifications Δ) and a GHD of the original hypergraph to the input.

SearchUpdateGHD(Δ)

Instance:	hypergraph H, modification $\delta \in \Delta$, a minimal width GHD of H
Output:	A GHD of $\delta(H)$ with width $\leq ghw(H)$ if it exists or answer 'no' otherwise.

In the future it might be of interest to further generalise the SearchUpdateGHD to decide whether a modification increases the width of H only up to some constant threshold c, i.e., whether $ghw(\delta(H)) \leq ghw(H) + c$. However, note that of the elementary modifications introduced above, only DelConstr can actually increase the ghw of the hypergraph by more than 1. Thus, for all introduced modifications except DelConstr this generalised problem is trivially true for any $c > 0$. For DelConstr the increase in width can depend fully on the structure that was "hidden" by the deleted constraint. We therefore focus on the analysis of SearchUpdateGHD in this paper and leave the generalised version open for future work on settings with more complex modifications.

Importantly, SearchUpdateGHD is a search problem rather than a decision problem. This is motivated from two sides. Our primary motivation stems from practical situation in which small, iterative updates are consistently made to some CSP and for which we want to maintain a low width GHD. Since the GHD is necessary to possibly exploit low width for solving the CSP, we are interested in the search problem rather than the decision problem.

The second motivating factor comes from the possibility of certain classes of modifications capturing other classes, i.e., if one can express some modification in a class Δ via a sequence of modifications from another class Δ'. Focusing purely on the decision problem makes it problematic to consider the complexity of sequences of updates, since we have no information on the complexity of obtaining the new input GHDs along the sequence of updates. By studying the search problem instead we can make strong statements for such cases.

The complexity of search problems (see [18]) is a complex topic and the full theoretical framework is not necessary in our context here. Instead, we will be content with showing that even for the simple classes of atomic updates that were discussed previously (except DelVar), SearchUpdateGHD can not be solved in polynomial time. Note that SearchUpdateGHD trivially reduces to the problem of finding an optimal GHD of $\delta(H)$ and all negative results therefore extend also to finding optimal GHDs under modifications.

Theorem 5.1. *For $\Delta \in \{$AddEq, DelEq, AddVar, AddConstr, DelConstr$\}$, SearchUpdateGHD(Δ) cannot be solved in polynomial time (assuming $\mathcal{P} \neq \mathcal{NP}$).*

Proof Idea. The basic strategy for each modification class Δ is simple. We show how to decide an $\mathcal{NP}$-hard decision problem by finding an initial hypergraph H_0, which can be modified by some sequence of $\delta_1, \delta_2, \dots, \delta_\ell \in \Delta$ to some target H. The decision problem will be equivalent to the question whether $ghw(H) \leq ghw(H_0)$. However, this strategy presents us with two technical challenges. First, the initial H_0 needs to be chosen in such a way that a minimal width GHD can be constructed in polynomial time. Second, there can be no index i such that after applying the first i modifications to H_0, we get a H_i with $ghw(H_0) < ghw(H_i)$ even when $ghw(H) \leq ghw(H_0)$. That is, the sequence can not increase the width at intermediate hypergraphs before decreasing again.

To tackle these issues, we do not reduce from ghw checking because the second challenge is particularly problematic. Indeed, our operations are not monotonic (w.r.t. ghw) in general. Instead, we reduce from 3-Sat by building on the proof of $\mathcal{NP}$-hardness of ghw checking (for constant width) given by Gottlob et al. [69]. There, a hypergraph is constructed that has ghw 2 exactly if some 3-Sat instance is satisfiable and has ghw 3 otherwise. Using this specific hypergraph we give concrete H_0 and modification sequences as described above for each Δ. We defer the full proof for each Δ to Section 5.5. $\square$

Updating GHDs is computationally difficult for all of the natural atomic operations that we considered, except for DelVar[1] (where the problem is trivial as DelVar cannot increase width and a new GHD is trivial to construct). As part of the proof of Theorem 5.1 we discuss how to decide 3-Sat via sequences of modifications, as long as those sequences adhere to certain conditions. Using this observation we can strengthen the statement from Theorem 5.1 to all modification classes that capture any of the hard atomic cases in the following formal sense.

For a sequence of modifications $\delta_1, \delta_2, \dots, \delta_\ell$ let us write $\delta_1^n(H)$ as a shorthand for $\delta_n(\delta_{n-1}(\cdots(\delta_1(H))\cdots))$. Let Δ, Δ' be two sets of modifications. We say that Δ *polynomially captures* Δ' if for every hypergraph H and $\delta' \in \Delta'$ there exists a sequence $\delta_1, \delta_2, \dots, \delta_\ell$ of modifications in Δ such that $ghw(\delta_1^i(H)) \leq ghw(\delta_1^{i+1}(H))$ for $1 \leq i < \ell$, $\delta_1^\ell(H) = \delta'(H)$ and ℓ is polynomially bounded in the size of H. In plain terms, every modification in Δ' can equivalently be reached via a polynomial sequence of modifications from Δ.

[1]A DelVar modification results in an induced subhypergraph, which is well known to never increase in width (see e.g., [69]).

Corollary 5.1. *Let Δ be a class of modifications that polynomially captures at least one class among* AddEq*,* DelEq*,* AddVar*,* AddConstr*, or* DelConstr*. Then* SearchUpdateGHD(Δ) *cannot be solved in polynomial time (assuming $\mathcal{P} \neq \mathcal{NP}$).*

5.2 A Framework for Handling Updates

We have seen that SearchUpdateGHD is difficult in general. In the following we thus focus on making the first steps towards practical solutions for the problem. In this section we present the theoretical framework of *mutable subtrees* for the uniform treatment of GHD updates under arbitrary modifications. Moreover, we briefly discuss how our approach extends to sequences of elementary modifications.

5.2.1 The δ-mutable Subtrees of a Decomposition

We lay the theoretical foundations of δ-mutable subtrees, a notion that will let us treat updates uniformly. We first introduce some convenient notation. Let $\langle T, (B_u)_{u \in T}, (\lambda_u)_{u \in T} \rangle$ be a GHD of a CSP P and let T' be a subtree of T. We write $T \setminus T'$ for the forest created by removing the nodes of T' from T. Since we are interested in the hypergraph structure, we write $H[T']$ for the subhypergraph of H induced by the vertices $\bigcup_{u \in T'} B_u$.

We are now ready to introduce the central notion of our framework, δ-mutable subtrees. Intuitively, these subtrees (of a decomposition) represent a kind of *local neighborhood* of the modification δ, i.e., the segment of the decomposition that corresponds to those parts of the hypergraph that are changed by δ. Note that the definitions and results in this section apply not only to the previously discussed elementary modifications but to arbitrary modifications in the sense of Definition 5.1.

Definition 5.5 (δ-mutable subtree)**.** *Let $\mathcal{G}$ be a GHD of hypergraph H with tree T, and let δ be a modification. A subtree T^* of T is a* δ-mutable subtree *if the following conditions hold:*

- $H[T \setminus T^*] = \delta(H)[T \setminus T^*]$,
- *and no $v \in V(\delta(H)) \setminus V(H)$ is adjacent (in $\delta(H)$) to a vertex in $B(T \setminus T^*)$.*

Thus, we split our existing decomposition in two parts: the mutable subtree T^*, where the corresponding part of the hypergraph has changed, and the *outer* subtrees which correspond to those subhypergraphs that remain unchanged by the modification. An important reason for considering mutable subtrees is captured by the following Lemma 5.1, namely that all the trees outside of T^* are still correct GHDs for their respective parts of the new hypergraph. Hence, it is possible to reuse these partial decompositions for $\delta(H)$ and save the effort of decomposing those parts of the hypergraph again.

Lemma 5.1. *Let $\langle T, (B_u)_{u\in T}, (\lambda_u)_{u\in T}\rangle$ be a GHD of hypergraph H with tree T, let δ be a modification, and let T^* be a δ-mutable subtree. For every tree T' in the forest $T \setminus T^*$ it holds that $\langle T', (B_u)_{u\in T'}, (\lambda_u)_{u\in T'}\rangle$ is a GHD of $\delta(H)[T']$.*[2]

Proof. Since we assume that T' is a tree in the forest $T \setminus T^*$ we also have that $B(T') \subseteq B(T \setminus T^*)$. Hence, it must also hold that $H[T'] = \delta(H)[T']$ by assumption that T^* is δ-mutable.

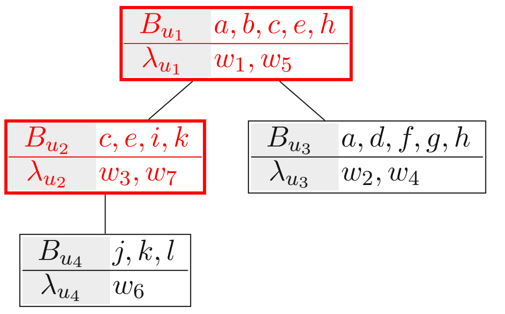
We now argue that $\langle T', (B_u)_{u\in T'}, (\lambda_u)_{u\in T'}\rangle$ is a GHD of $H[T']$ and thus, by the previous argument, also of $\delta(H)[T']$. First, observe that the connectedness condition is clearly still satisfied in T' since we never change the bags. For the covers it is clear that if $e \in E(H)$, then $e \cap B(T')$ is an edge in $H[T']$. Since $B_u \subseteq B(T')$ we clearly also have that $B_u \subseteq \bigcup_{e\in\lambda_u} e \cap B(T')$. What is left, is to verify that every edge e of $H[T']$ is covered in T'. Let e' be one of the edges in H such that $e = e' \cap B(T')$. Since we start from a GHD of H, there must be a node u where e' is covered. Hence, all the vertices of e are in B_u. Hence, all of the subtrees induced by the vertices in e touch by the connectedness condition. Since all of the vertices of e are in $B(T')$ all those subtrees must have a common node in T'. Hence, $\langle T', (B_u)_{u\in T'}, (\lambda_u)_{u\in T'}\rangle$ is a GHD of $H[T']$ and therefore also of $\delta(H)[T']$. ☐

Example 5.3. *In Example 5.2, a $\delta \in$ AddConstr is used to create the hypergraph H_{P_2} from $H_{P'}$, as in Figure 5.3a. We now consider reverting this modification, i.e., the modification $\delta^{-1} \in$ DelConstr that removes the edge $\{c, i\}$, i.e., we have $\delta^{-1}(H_{P_2}) = H_{P'}$ (recall, the hypergraph of P' is shown in Figure 5.2b). As input for our update example, we use the width 2 GHD $\langle T, (B_u)_{u\in T}, (\lambda_u)_{u\in T}\rangle$ of H_{P_2} given in Figure 5.4a. The two highlighted nodes in Figure 5.4a represent a δ^{-1}-mutable subtree T^* of T. Observe that $T \setminus T^*$ consists of two trees that correspond to the induced subhypergraphs in Figure 5.4b. By Lemma 5.1, these parts remain correct GHDs for their respective induced subhypergraphs.*

We could update the overall decomposition by changing the bag $\{c, e, i, k\}$ to $\{e, i, k\}$ while removing w_7 from the λ label to update the decomposition to fit P'. Mechanically this can be checked by searching for a GHD of $\delta^{-1}(H_{P_2})[\{a, b, c, e, i, h, k\}]$ that is consistent with the surrounding trees in a certain way that will be discussed below.

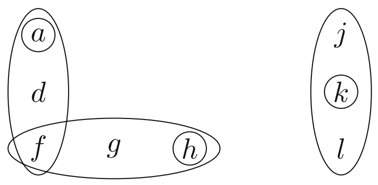
Since we want to reuse as much of the old decomposition as possible, it naturally becomes interesting to have $T \setminus T^*$ as large as possible. Hence, we are interested in finding *minimal* δ-mutable subtrees, i.e., those δ-mutable subtrees with the least number of nodes. Fortunately, it is relatively easy to find minimal mutable subtrees. The full tree T is trivially a δ-mutable subtree. We can then start from $T_0 = T$ and greedily eliminate leaves as long as the property from Definition 5.5 remains valid. Once no more leaves can be removed, the procedure will have reached a minimal δ-mutable subtree.

[2]Technically every edge e in every edge cover λ_u is replaced by the edge $e \cap (\bigcup_{u\in T'} B_u)$ of the induced subhypergraph.



B_{u_1} a, b, c, e, h
λ_{u_1} w_1, w_5
B_{u_2} c, e, i, k
λ_{u_2} w_3, w_7
B_{u_3} a, d, f, g, h
λ_{u_3} w_2, w_4
B_{u_4} j, k, l
λ_{u_4} w_6

(a) The bags of a GHD of H' with width 2. The minimal δ-mutable subtree is highlighted.



(b) Hgs induced by $T \setminus T^*$.

Figure 5.4: Example 5.3.

Lemma 5.2. *For any GHD $\mathcal{G}$ of a hypergraph H and any modification δ there exists a* unique *minimal δ-mutable subtree. Moreover, there exists an algorithm with input $(\mathcal{G}, H, \delta(H))$ that computes the minimal δ-mutable subtree in polynomial time.*

Proof. We first prove the uniqueness of minimal δ-mutable subtrees. Suppose towards a contradiction that there are two distinct minimal δ-mutable subtrees T_1 and T_2 of a GHD $\langle T, (B_u)_{u \in T}, (\lambda_u)_{u \in T} \rangle$. Recall, in the argument for Lemma 5.1 it was already argued that for every tree T' in $T \setminus T_1$ or $T \setminus T_2$, we have that $H[T'] = \delta(H)[T']$.

Now, from the assumption that both T_1 and T_2 are minimal but distinct there has to exist a tree $T' \in T \setminus T_1$ such that $T' \cap T_2 \neq \emptyset$. If this were not true, then T_2 would be a subtree of T_1 and we are done. Fix such a T' and let X be the set of nodes that are in T' and T_2. Since $H[T'] = \delta(H)[T']$, and $X \subseteq T'$, also $H[X] = \delta(H)[X]$ and it becomes easy to see that

$$H[T \setminus (T_2 \setminus X)] = H[B(X) \cup B(T \setminus T_2)] = \delta(H)[B(X) \cup B(T \setminus T_2)] = \delta(H)[T \setminus (T_2 \setminus X)]$$

Thus, $T_2 \setminus X$ is also a δ-mutable subtree and smaller than T_2, contradicting our initial assumption of minimality. Note that the second condition of Definition 5.5 cannot become unsatisfied by removing nodes from T_2.

The algorithm from the statement is given in Algorithm 5.1. The algorithm clearly starts with a δ-mutable subtree and throughout the iterative elimination the *working tree* T' remains a δ-mutable subtree. Note that the argument from before can be seen as a method to create a smaller δ-mutable subtree from any disjoint pair of δ-mutable subtrees. Hence, if no more leaves can be removed from T' in the algorithm, then every smaller tree must not be disjoint. But if the minimal subtree were to be a proper subtree of T', then removal of some leaf must be possible since the induced subhypergraphs of $T \setminus T'$ grow monotonically. Thus, there can be no smaller δ-mutable subtree that is a subtree of T' and none that has disjoint vertices from T'. It follows that the returned T' is minimal and the algorithm is therefore correct. $\square$

Algorithm 5.1: Finding Minimal δ-mutable Subtrees.

```
Input: Hypergraph $H$, Hypergraph $\delta(H)$, GHD $\langle T, (B_u)_{u \in T}, (\lambda_u)_{u \in T} \rangle$ of $H$
Output: A minimal $\delta$-mutable subtree $T'$
begin
    $T' \leftarrow T$
    $V_{new} \leftarrow V(\delta(H)) \setminus V(H)$
    repeat
        foreach Leaf $u$ of $T'$ do
            $T'_{-u} \leftarrow T' \setminus \{u\}$
            $A \leftarrow H[T \setminus T'_{-u}]$
            $B \leftarrow \delta(H)[T \setminus T'_{-u}]$
            $Adj \leftarrow$ all vertices adjacent to $B(T \setminus T'_{-u})$ in $\delta(H)$
            if $A = B$ and $V_{new} \cap Adj = \emptyset$ then
                $T' \leftarrow T'_{-u}$
                break
            end
        end
    until $T'$ did not change
    return $T'$
end
```

5.2.2 Updating GHDs using δ-mutable Subtrees

By Lemma 5.1 we can use the old decomposition to derive correct GHDs for certain induced subgraphs of $\delta(H)$. It is not guaranteed that the minimal width GHD of $\delta(H)$ can be constructed in such a way that these pre-solved induced subgraphs correspond to parts of the decomposition. However, the possibility of only having to recompute a decomposition for some small subgraph $\delta(H)^*$ is promising in practice. In particular, we are interested in $\delta(H)^*$ which is the part of $\delta(H)$ that contains $\delta(H)[T^*]$ for the minimal δ-mutable subtree T^*, plus any possible new vertices and edges introduced by δ. Ideally, we want a new GHD for $\delta(H)^*$ with which we can replace $T^* \subseteq T$ to arrive at a valid generalized hypertree for $\delta(H)$. This way we can fully reuse the $T \setminus T^*$ parts of the old GHD. To replace the new decomposition of $\delta(H)^*$ in place of T^* in T we need to enforce some additional constraints on the GHD of H^*. Therefore, we introduce the notion of *bag constraint* as a set $\gamma \subseteq V(\delta(H)^*)$. A bag constraint γ is satisfied by a GHD $\langle T, (B_u)_{u \in T}, (\lambda_u)_{u \in T} \rangle$ if there exists a node $u \in T$, s.t., $\gamma \subseteq B_u$. In particular, given such a GHD and a mutable subtree T^*, let $\{u_1, \dots, u_q\}$ be the set of nodes in $T \setminus T^*$ that have a neighbor in T^*. We call the set $\{\gamma_i \mid \gamma_i = B_{u_i} \cap (\bigcup_{u \in T^*} B_u), 1 \leq i \leq q\}$ the *T^*-induced bag constraints.*

Theorem 5.2. *Let $\mathcal{G}$ be a width k GHD of a hypergraph H with tree T, let δ be a modification and let T^* be a δ-mutable subtree of T. If $\delta(H)^*$ has a GHD of width $\leq k$*

that satisfies all T^*-induced bag constraints, then $ghw(\delta(H)) \leq k$.

Proof. We prove the statement by constructing the required new width k GHD of $\delta(H)$ from the GHD of $\delta(H)^*$ and the subtrees $T \setminus T^*$. Hence, not only is the width of $\delta(H)$ at most k, but a GHD of $\delta(H)$ can be efficiently constructed by only computing a GHD (with bag constraints) for $\delta(H)^*$.

Suppose $\mathcal{D}^*$ is a width k GHD of H^* that satisfies all T^*-induced bag constraints. Let $\gamma_1, \dots, \gamma_\ell$ be T^*-induced bag constraints and recall that every bag constraint is associated one-to-one to a node in T that neighbors a node in T^*. Let u_i be the node associated to the constraint γ_i in this way for all $i \in [\ell]$.

The final decomposition $\langle T', (B'_u)_{u \in T'}, (\lambda'_u)_{u \in T'} \rangle$ is now constructed as follows starting from $\mathcal{D}^*$. For each bag constraint γ_i, identify the subtree $T_i \in T \setminus T^*$ that contains u_i as well as any node u_i^* in $\mathcal{D}^*$ that satisfies γ_i. Then, attach the tree T_i at node u_i to $\mathcal{D}^*$ at u_i^*. By attaching subtrees for each bag constraint this way we obtain our final $\langle T', (B'_u)_{u \in T'}, (\lambda'_u)_{u \in T'} \rangle$.

We now argue that $\langle T', (B'_u)_{u \in T'}, (\lambda'_u)_{u \in T'} \rangle$ is indeed a width k GHD of $\delta(H)$. Indeed, width k follows immediately from the construction since $\mathcal{D}^*$ and $\langle T, (B_u)_{u \in T}, (\lambda_u)_{u \in T} \rangle$ both have width k and none of their λ-labels are modified. For connectedness, recall that by our definition of bag constraints the tree T_i is attached to a node u_i^* whose bag contains $B_{u_i} \cap B(T^*)$. Hence, every vertex in $B(T^*)$, and thus also every vertex in bags of $\mathcal{D}^*$, that also occurs in $B(T_i)$ must be in B_{u_i}. We see that connectedness can not be violated by the attaching step of our construction. By Lemma 5.1, all the individual parts that are attached to $\mathcal{D}^*$ already satisfy the connectedness condition and it therefore holds also for all of $\langle T', (B'_u)_{u \in T'}, (\lambda'_u)_{u \in T'} \rangle$.

Finally, we verify that all edges of $\delta(H)$ are covered by some bag of $\langle T', (B'_u)_{u \in T'}, (\lambda'_u)_{u \in T'} \rangle$. We partition the set of edges in two sets, edges that are in $\delta(H)^*$ and those that are not. If an edge is in $\delta(H)^*$, then it must be covered by $\mathcal{D}^*$ and thus also in $\langle T', (B'_u)_{u \in T'}, (\lambda'_u)_{u \in T'} \rangle$. In the latter case, observe that if an edge e is in $\delta(H)$ but not in $\delta(H^*)$, then e is in H and thus covered by some node of $T \setminus T^*$. Note that there is a T^*-induced bag constraint for every tree in $T \setminus T^*$. Hence, by Lemma 5.1 and the above construction reattaching the subtree in which e is covered, e is also covered in $\langle T', (B'_u)_{u \in T'}, (\lambda'_u)_{u \in T'} \rangle$.

Note that we made no explicit use of the second condition in Definition 5.5. The condition effectively enforces that any edges that contain new vertices will be in H^* and in this way implicitly factors into the above argument. □

It is possible for no T^*-induced bag constraints satisfying GHD of $\delta(H)^*$ with width at most k to exist, even if $ghw(\delta(H)) \leq k$. Thus, while the discussions of this section – and in particular the ideas of Theorem 5.2 – form the foundation of our practical implementation, some adaptations are necessary to efficiently deal with those cases. This will be the topic of the following section.

5.3 Implementation of the δ-mutable Subtree Framework

We now focus on using the concept of δ-mutable subtrees to update GHDs systematically. The idea is to speed up the computation of a new GHD for an updated hypergraph by exploiting information that can be inferred from an old GHD of the original instance, even if parts of the old GHD need to be recomputed. We achieve this by devising a data structure that can then be maintained throughout a sequence of successive updates to speed up the computation even further. We define an algorithmic framework that is implementation agnostic in the sense that only a few adaptations are required to extend the capabilities of handling GHD updates to virtually any existing top-down algorithm computing GHDs for the classical case. While here we just provide the abstract version of this framework, in Section 5.4 we implement it into an existing state-of-the-art decomposition algorithm. There we also report on its performance in various update tasks.

5.3.1 Reusable Subtrees and Scene Mappings

The goal of our implementation is twofold: we want a strategy built on top of the framework of δ-mutable subtrees, and we want it to encompass existing algorithms for computing GHDs. In the following, we make use of the basics of top-down GHD construction, as explained in Section 2.4. Before we proceed with explaining our implementation, we introduce a way of referring to how bags and edge covers of the old GHD are affected by an update δ when we want to use the old GHD with the modified hypergraph $\delta(H)$. We first define a function $s_\delta : E(H) \to E(\delta(H)) \cup \emptyset$, which maps edges $e \in E(H)$ to their corresponding equivalent $e' \in E(\delta(H))$, if it exists, or $\emptyset$, if δ actually deleted that edge. By slight abuse of notation, given a subset $X \subseteq E(H)$, we shall use $\delta(X) = \{s_\delta(e) \mid e \in X\}$. In this same vein, we also introduce for a vertex set $Y \subseteq V(H)$, the notation $\delta(Y) = Y \cap V(\delta(H))$. Another notational choice we make throughout this section is how to refer to the input of an algorithm dealing with updated hypergraphs. Since all algorithms we present only deal with a single updated hypergraph and its subgraphs and never need the original hypergraph, we omit the use of the δ function for hypergraphs. So instead of $\delta(H)$, we just write H, with the understanding that H has already been updated. We will still need a δ-mutable subtree, but we assume that it has been computed and provided to the algorithm as an input.

The idea behind our framework is to try to update the minimal δ-mutable subtree T^* and reuse as many of the *outer* subtrees as possible. If this is not possible, due to the way the modification has changed the hypergraph, we still want to return a GHD of the updated hypergraph quickly. Bag constraints from Theorem 5.2 encode the properties necessary for parts of T^* to be reused. As mentioned, however, it is possible that in order to successfully find a new GHD of low width, we need to forgo some of them. For our implementation we think of them as *soft constraints*: we make an effort to find GHDs that reuse T^* if they exist, and if they do not, use them as a starting point in the search space. We realize this behavior via the concept of a *scene*.

Definition 5.6 (Scene)**.** *Let $\langle T, (B_u)_{u \in T}, (\lambda_u)_{u \in T} \rangle$ be a normal-form GHD of a hypergraph H. A scene mapping $\sigma \colon 2^{E(H)} \to N(T)$ is a partial mapping from a subhypergraph $H' \subseteq H$ to a node $u \in T$. The co-domain element of σ is denoted as a* scene*. Given a modification δ and a δ-mutable subtree T^*, we call $\sigma(H)$* out-scene *if $\sigma(H) \notin T^*$ or* in-scene *if $\sigma(H) \in T^*$.*

Scenes are used to avoid decomposing again parts of the hypergraph for which we already know a GHD. Lemma 5.1 implies that out-scenes are reusable for a new GHD. Using in-scenes is more complex, thus, we try to utilize them at most once to see if they help in finding a GHD of the updated instance. If this leads to a reject case, we know that the scene will not be used again. Therefore, in-scenes do not invalidate the correctness of our approach. We compute a scene mapping via a two-phase traversal of the old GHD $\mathcal{G}$. Without loss of generality (see Theorem 2.1), we require $\mathcal{G}$ to be in normal form so that we can determine which subtrees of the GHD "cover" certain components of the new hypergraph H. Recall that this step can be performed in polynomial time [72]. Additionally to $\mathcal{G}$ and H, we also need to know which nodes of $\mathcal{G}$ belong to the δ-mutable subtree of H. We assume that this tree has been computed prior to the start of the computation of the scene mapping and is referred to as T^*. The pseudocode of our procedure is split into Algorithm 5.2 and Algorithm 5.3, each detailing one of the two phases. We proceed to give an informal explanation below.

The *downward phase*, called `SceneCreationDown`, is executed first. In Algorithm 5.2, the old GHD is traversed top-down as the bags of the nodes encountered along the traversal are used to "replay a decomposition procedure". The first call of the algorithm is performed on the root n of the old GHD, the updated hypergraph H, and the δ-mutable subtree T^*. In lines 2–9, we check whether the current node n trivially cannot be reused for a new decomposition of H. In this case, we proceed with the second phase of our procedure. Otherwise, if the previous checks are passed, we create a new mapping $H \to n$ for the current hypergraph H as seen in line 10. The next step consists in computing a mapping between the $[\delta(B_n)]$-components $C_1, \ldots, C_\ell$ of H and the child nodes $u_1, \ldots, u_\ell$ of n (line 11). Here we assign each child u_i to a component C_i where we have that $\delta(B_{u_i}) \cap C_i \neq \emptyset$. Due to the properties of GHDs in normal form, we know that each $[\delta(B_n)]$-component matches with exactly one child node u_i. If not, then the downward phase stops jumping to line 13. This can only happen when considering nodes of T^*. However, if a mapping exists and is unique, we make a recursive call on each pair (u_i, C_i) in line 16 and subsequently collect the returned scenes. Finally, the resulting scene mapping σ is returned.

The *upward phase*, called `SceneCreationUp`, is started by Algorithm 5.2 anytime this stops at a non-leaf node, as seen in lines 3, 8 and 13. While traversing the remaining nodes of the old GHD, Algorithm 5.3 skips all those nodes belonging to T^* and tries to produce new scenes for any subtree T_n attached below T^*. This is done by recursively calling the function on the children of n until a leaf is reached, as seen between lines 3 – 7. Since `SceneCreationUp` is called only on nodes $n \in T$ such that $n \notin T^*$ and T^* is

Algorithm 5.2: The `SceneCreationDown` Algorithm.

Input: Node $n \in T$, Hypergraph H, δ-mutable subtree T^*, Modification δ
Output: A scene mapping σ

```
1  begin
2      if $\delta(B_n) \nsubseteq B(\delta(\lambda_n))$ then
3          return SceneCreationUp($n$, $H$, $T^*$)
4      end
5      $\mathcal{X} \leftarrow [\delta(B_n)]$-components of $H$
6      $\mathcal{Y} \leftarrow$ children of $n$
7      if $|\mathcal{X}| \neq |\mathcal{Y}|$ then
8          return SceneCreationUp($n$, $H$, $T^*$)
9      end
10     $\sigma \leftarrow \{H \to n\}$
11     compute mapping $\theta$ such that $\forall u, C \in \mathcal{Y} \times \mathcal{X}$, $\theta(u) = C$ iff
         $V(C) \setminus (\delta(B_n) \cap \delta(B_u)) \neq \emptyset$
12     if $\theta$ does not exists or $\theta$ is not unique then
13         return SceneCreationUp($n$, $H$, $T^*$)
14     end
15     foreach $u, \theta(u) \in \theta$ do
16         $\sigma' \leftarrow$ SceneCreationDown($u$, $\theta(u)$, $T^*$)
17         $\sigma \leftarrow \sigma \cup \sigma'$
18     end
19     return $\sigma$
20 end
```

a tree, hence a connected set of nodes, the check in line 8 is sufficient to skip all and only those nodes belonging to T^*. At this point, the algorithm creates new scenes by going back through the calls in a bottom-up fashion and at every node n it looks at the subtree T_n to create the mapping $\{e \in E(H) \mid e \cap B(T_n)\} \to n$. To this end, the edges of H "covered" by the subtrees rooted in the children of T_n are collected through the calls (line 6). The algorithm proceeds by computing the edges covered by the bag of n in line 9. The union of this set of edges with $\{e \in E(H) \mid e \cap B(T_n)\}$ represent all the edges constituting $H[T_n]$. We thus map $H[T_n]$ to n, as seen in line 12. This upward phase ensures we can make full use of Lemma 5.1 by considering all subtrees below T^* that are not affected by the modification at hand.

Example 5.4. *We shall consider here as our initial hypergraph H_{P_2}, seen in Figure 5.3. A GHD of H_{P_2} is provided in Figure 5.4a, we shall refer to it as $\mathcal{G}$ in the sequel. We will use the same modification $\delta^{-1} \in$ DELCONSTR as introduced in Example 5.3. Thus, using $\delta^{-1}(H_{P_2})$ and $\mathcal{G}$ we will create the scene mapping. We start with the downward phase.*

Looking at the root node u_1 of $\mathcal{G}$, we create a scene mapping $\delta^{-1}(H_{P_2}) \to u_1$. Next, we

Algorithm 5.3: The `SceneCreationUp` Algorithm.

```
Input: Node $n \in T$, Hypergraph $H$, $\delta$-mutable subtree $T^*$
Output: A scene mapping $\sigma$, a hypergraph $H[T_n]$
begin
    $\mathcal{Y} \leftarrow$ children of $n$
    for $u \in \mathcal{Y}$ do
        $\sigma', H[T_u] \leftarrow$ SceneCreationUp $(u, H, T^*)$
        $\sigma \leftarrow \sigma \cup \sigma'$
        $H[T_n] \leftarrow H[T_n] \cup H[T_u]$
    end
    if $n \notin T^*$ then
        $H[T_n] \leftarrow H[T_n] \cup \{e \in E(H) \mid e \subseteq B_n\}$
        $\mathcal{X} \leftarrow [B_n]$-components of $H[T_n]$
        if $|\mathcal{X}| = |\mathcal{Y}|$ then
            $\sigma \leftarrow \sigma \cup \{H[T_n] \rightarrow n\}$
        end
    end
    return $\sigma, H[T_n]$
end
```

consider the $[B_{u_1}]$-components of $\delta^{-1}(H_{P_2})$, yielding components, $C_2 = \{w_3, w_6\}$ and $C_3 = \{w_2, w_4\}$. We look for unique matching pairings of child nodes of u_1 and $[B_{u_1}]$-components. We see that $(B_{u_2} \setminus B_{u_1}) \cap V(C_2) = \{i, k\}$ and $(B_{u_3} \setminus B_{u_1}) \cap V(C_3) = \{d, f, g\}$. Since all components were matched, we proceed on the pairings (C_2, u_2) and (C_3, u_3). Next we consider the node u_2, and create the mapping $C_2 \rightarrow u_2$. We consider now the $[B_{u_2}]$-components of C_2. We get one component, $C_4 = \{w_6\}$ and we have that $(B_{u_4} \setminus B_{u_2}) \cap V(C_4) = \{j, l\}$. Thus we proceed on the pairing (C_4, u_4). We create the mapping $C_4 \rightarrow u_4$. We note that there are no $[B_{u_4}]$-components of C_4, since B_{u_4} already covers the entire component C_4. We continue with (C_3, u_3). We create the mapping $C_3 \rightarrow u_3$. As before, we note that there are no $[B_{u_3}]$-components of C_3, as B_{u_3} already fully covers C_3. Since the downward phase never stopped at a non-leaf node, we do not proceed to the upward phase. To summarize, we get the following scene mapping: $\{(\delta^{-1}(H_{P_2}) \rightarrow u_1), (C_2 \rightarrow u_2), (C_3 \rightarrow u_3), (C_4 \rightarrow u_4)\}$. We will see in Algorithm 5.4 how this scene mapping can be used to speed up GHD computation under updates.

Lemma 5.3. *The `SceneCreationUp` function, detailed in Algorithm 5.3, has a time complexity of $O(N^2 \cdot E \cdot V)$ where N is the number of nodes of the input tree T, and V and E are the number of vertices and edges of the input hypergraph H, respectively.*

Proof. Here we assume that the nodes of T belonging to T^* have been explicitly marked and, thus, for any $n \in T$, checking whether $n \in T^*$ or not takes constant time.

Let us first consider a call of Algorithm 5.3 on a leaf node of $n \in T$ such that $n \notin T^*$. Hence, lines 3-7 are skipped, while lines 8-13 are all executed. The first salient operation of this call is the construction of the current induced subhypergraph $H[T_n]$. Here, the computation of the set $\{e \in E(H) \mid e \subseteq B_n\}$ roughly costs $O(E \cdot V \cdot N)$ by subset checking. The next relevant operation is the computation of the connected components of $H[T_n]$ separated by B_n. The algorithm from [129] computing the connected components of graphs can be adapted to the case of hypergraphs[3], thus contributing with a $O(V+E)$ cost. Then, the check in line 10 requires counting the number of components $\mathcal{X}$ and the number of children $\mathcal{Y}$ of n, the cost of which can be overestimated with $O(V+N)$. The cost of such a call is thus dominated by $O(E \cdot V \cdot N)$.

Conversely, in a general call to `SceneCreationUp`, lines 3-7 execute only constant time operations except for the recursive call to `SceneCreationUp`. It is easy to see that `SceneCreationUp` is executed exactly once per each node of T_n. Therefore, the total cost of Algorithm 5.3 is $O(N \cdot (E \cdot V \cdot N))$. □

Lemma 5.4. *The `SceneCreationDown` function, detailed in Algorithm 5.2, has a time complexity of $O(N^2 \cdot E \cdot V + N^2 \cdot E^2 + N \cdot V^2)$ where N is the number of nodes of the input tree T, and V and E are the number of vertices and edges of the input hypergraph H, respectively.*

Proof. Let us first consider the case of a run of Algorithm 5.2 that never makes a subcall to Algorithm 5.3. Here, it is easy to see that `SceneCreationDown` is executed once per each node of the input tree T and it is thus called N times. Therefore, the total cost of such a run is $N \cdot cost($`SceneCreationDown`$)$. We shall now calculate the cost of executing lines 2-11. Assuming that $\delta(B_n)$ and $\delta(\lambda_n)$ are part of the input, checking if $\delta(B_n) \nsubseteq B(\delta(\lambda_n))$ costs $O(V^2)$ since these are both subsets of $V(H)$. We then proceed to the computation of the $[\delta(B_n)]$-components of H, which costs $O(V+E)$ as discussed in the complexity proof of `SceneCreationUp`. The next check $|\mathcal{X}| \neq |\mathcal{Y}|$ costs $O(E+N)$. The computation of the mapping between children of the current node n and $[\delta(B_n)]$-components of H can be performed by comparing the $N \cdot E$ pairs of elements from $\mathcal{Y}$ and $\mathcal{X}$. For each pair, we then check whether the condition $V(C) \setminus (\delta(B_n) \cap \delta(B_u)) \neq \emptyset$ holds or not, which can be performed in $O(E+2V)$ time. The total cost of the mapping computation is thus $O(N \cdot E \cdot (E+2V))$. Finally, the total cost of a run of `SceneCreationDown` is $O(N \cdot (V^2 + (V+E) + (E+N) + N \cdot E \cdot (E+2V))) = O(N \cdot (N \cdot E \cdot V + N \cdot E^2 + V^2))$.

Let us now take into account the possible calls to `SceneCreationUp`. We observe that for each node $n \in T$ either `SceneCreationDown` of `SceneCreationUp` is called unless n is a "frontier" node, i.e., a node where one of the checks in `SceneCreationDown` fails and `SceneCreationUp` is called. Nevertheless, once `SceneCreationUp` starts, the recursion of `SceneCreationDown` ends. It is thus reasonable to overestimate the general cost of Algorithm 5.2 with the cost of a call to `SceneCreationDown` where

[3]While the cited article deals with (strongly) connected components of directed graphs, a trivial reduction to the undirected setting exists. We can then consider a hypergraph as an undirected graph.

`SceneCreationUp` is never called plus the cost of call to `SceneCreationUp` on the original input of Algorithm 5.2. Then, the total cost is $O(N^2 \cdot E \cdot V + N^2 \cdot E^2 + N \cdot V^2)$. □

We conclude this section with a final note on the complexity of transforming an existing GHD into normal form. Algorithm 5.2 and Algorithm 5.3 require that the input GHD is in normal form. Theorem 5.4 in [72] shows a polynomial-time algorithm to perform this transformation on HDs, but it can straightforwardly be applied to GHDs as well. Intuitively, given a hypergraph H and a GHD T, this procedure visits every node n of the GHD and checks whether the conditions (5), (6), and (7) enlisted in Section 2.2 are satisfied or not. If not, the $[B_n]$-components of the subtree are computed and separated from one another by creating a distinct subtree for each component. The cost of these operations is roughly $O(N \cdot (V + E))$, where N is the number of nodes of T, while V and E are the number of vertices and edges of H, respectively.

5.3.2 Practical Recomputation of GHDs

Algorithm 5.4 is a pseudo-code representation of our framework. As input we expect four items:

1. a GHD $\mathcal{G}$ of the original hypergraph,
2. the updated hypergraph H,
3. the δ-mutable subtree T^*, and lastly,
4. a decomposition algorithm $\mathcal{D}$, which we call *decomposer*.

The output is a GHD of H of width $\leq k$, or a reject if none can be found. The decomposer $\mathcal{D}$ takes as input a hypergraph and a scene mapping and it produces a GHD of H of width $\leq k$, or rejects if none exists. Our algorithm initially computes a scene mapping σ, in line 15, using the aforementioned procedures. Then, the recursive function `DecompUpdate` is called on H and σ. At line 2, the function checks if a scene $\sigma(H')$ exists for the current subhypergraph H'. Here it is important to distinguish between out-scenes and in-scenes. Indeed, while the first can be reused without any restriction, the latter are used only once. We can thus think of this check as a stateful operation that changes the contents of σ in the following way: in-scenes will be removed from σ after the first time they have been checked and returned; for out-scenes, no such removal takes place. If the check has succeeded, then at line 3, the algorithm immediately fixes the current node of the GHD with $\sigma(H')$ and avoids the use of the decomposer, which would start an expensive search for a new bag. At line 4, we separate H' into the same $[B_u]$-components we encountered while computing the old GHD. Now we make a recursive call on each of these components in lines 6 to 9, adding each freshly computed GHD to the set of children of u. We then return the resulting GHD with u as its root. Line 12 is executed only if H' has never been encountered while building the old GHD. In this case, the decomposer $\mathcal{D}$ is called to find a GHD of H' of width $\leq k$.

Algorithm 5.4: GHD Recomputation with Scene Mappings.

```
Input: GHD $\mathcal{G}$, Hypergraph $H$, $\delta$-mutable subtree $T^*$, Decomposer $\mathcal{D}$
Output: A GHD of $H$ with width $\leq k$, or Nil if none exists
Parameter : Integer $k$
Function DecompUpdate($H'$: Hypergraph, $\sigma$: Scene Mapping)
    if $\sigma(H')$ is out-scene or $\sigma(H')$ not yet used then
        $u \leftarrow \sigma(H')$
        $\mathcal{X} \leftarrow [B_u]$-components of $H'$
        $u$.Children $\leftarrow \emptyset$
        for $c \in \mathcal{X}$ do
            $\mathcal{Y} \leftarrow$ DecompUpdate($c$, $\sigma$)
            $u$.Children $\leftarrow u$.Children $\cup \{\mathcal{Y}\}$
        end
        return $u$
    end
    return $\mathcal{D}$($H'$, $\sigma$)
end
begin /* Main */
    $\sigma \leftarrow$ SceneCreationDown(Root $r$ of $\mathcal{G}$, $H$, $T^*$)
    return DecompUpdate($H$, $\sigma$)
end
```

This design ensures that in either case, whether the δ-mutable subtree can be simply updated, or an entirely new GHD needs to be computed, we can use the same strategy. Moreover, in both cases we exploit the information provided by the old GHD. The decomposer itself can be any existing GHD algorithm, which just needs to be adapted to make use of scene mappings.

To demonstrate that our framework can be applied to existing combinatorial algorithms for finding GHDs, we decided to modify `BalancedGo` [74] to make use of the δ-mutable framework. Our extension supporting GHD updates is available at `https://github.com/dmlongo/BalancedGoUpdate`. It is notable that the original code had to be modified only in a few lines to fit the model of our decomposer $\mathcal{D}$ handling scene mappings, even though additional code had to be written for extracting the δ-mutable subtree from a given decomposition with respect to an updated hypergraph and for creating the scene mapping. The actual use of the scene mapping inside the general algorithm is straight-forward and we believe that almost any existing or future approach computing GHDs via a combinatorial process can make use of it.

Table 5.1: A breakdown of the instances used for the empirical evaluation by their widths prior to modification. In addition to the number of original instances, we also show the number of updated hypergraphs that originated from the relative original hypergraphs.

Category	Hypergraphs of Width									Total
	2	3	4	5	6	7	8	9	10	
Original	95	232	325	354	355	254	117	49	17	1798
Updated	2375	5800	8125	8850	8875	6350	2925	1225	425	44950

5.4 Empirical Evaluation

We now explore the potential of updating GHDs with the methods of Section 5.3. We describe our experiments, show their results and discuss the implications of our findings.

5.4.1 Methodology & Synthetic Update Generation

We compared multiple approaches for updating GHDs upon elementary modifications: *Update*, *ClassicGo*, and *ClassicLeo*. *Update* consists in our implementation of the general strategy of Section 5.3 built on top of the `BalancedGo` program from [74]. *ClassicGo* uses the original `BalancedGo` program to compute a GHD of the modified hypergraph from scratch. Finally, *ClassicLeo* uses `htdLEO` from [126] to similarly compute a GHD for the modified hypergraph from scratch.

More precisely, given a hypergraph H and a GHD $\mathcal{G}$ of H of width k, we applied an elementary modification δ to H and compared the times taken by *Update*, *ClassicGo* and *ClassicLeo* to output a GHD of $\delta(H)$ of width $\leq k$, if it exists. Recall that *Update* first computes the minimal δ-mutable subtree T^* from $\mathcal{G}$ and then tries to build a GHD of $\delta(H)$ of width $\leq k$ by reusing the parts of $\mathcal{G}$ that were not affected by δ.

We conducted our experiments on the HyperBench dataset [52]. HyperBench is a large collection of hypergraphs from applications, benchmarks, and random generation that has been successfully used in a variety of hypergraph decomposition experiments. By using the `LocalBIP` implementation of `BalancedGo` and the results of [74], we determined the optimal *ghw* of 1798 out of the 2035 CSPs of HyperBench within a timeout of 1 hour per instance. Indeed, updating a GHD of optimal width is the hardest case. We thus used these 1798 hypergraphs, their GHDs, and their *ghw* as a basis for our experiments.

For each hypergraph H, we randomly generated five elementary modifications per each class from Section 5.1.1 as follows. For AddVar we introduce a new vertex into ℓ randomly chosen edges, where ℓ is the average (rounded up) degree of the original hypergraph. We generate AddEq modifications by merging two random vertices and DelEq modifications by splitting a vertex x into two vertices y_1, y_2: in half of the edges incident to x, we replace x by y_1 and in the other half we replaced x by y_2. Notably, AddConstr adds an edge with average (rounded up) rank such that all vertices in the new edge are already part of some existing edge. That means that we generate

Table 5.2: Statistics for *ClassicGo*, *Update*, and *ClassicLeo* shown separately for each modification. *Mean ClassicGo*, *Mean Update*, and *Mean ClassicLeo* are in milliseconds. All mean times were rounded to the closest integer, all other non-integer numbers were rounded to two decimal places. Timeout was set to 30 minutes.

Operation	Positive (%)	Better (%)	Mean ClassicGo	Mean Update	Mean ClassicLeo	Mean Speedup	Timeout ClassicGo	Timeout Update	Timeout ClassicLeo
AddConstr	85.07	81.23	1106	27	55027	40.28	1269	757	3657
DelConstr	99.48	88.12	553	10	60826	53.78	534	105	3480
DelEq	95.59	73.2	714	82	58177	8.66	776	386	3677
AddEq	90.34	85.67	534	12.52	69843	42.68	675	336	3268
AddVar	85.76	65.73	1339	223	64166	5.99	1330	952	3807
Total	91.26	78.8	795	37	61239	21.77	4584	2536	17889

challenging cases while avoiding the easy case where most vertices in the new edge have no effect. For DelConstr a random edge is removed from the hypergraph. Note that updating a GHD of optimal width in case of $\delta \in$ DelVar is trivial. Indeed, let $v \in V(H)$ be the vertex removed by δ and consider a GHD $\mathcal{G}$ of H of width k. A GHD of $\delta(H)$ of width $\leq k$ can be easily obtained by removing v from all bags B_u of $\mathcal{G}$. In total, this process produces 44950 instances, each consisting of the original hypergraph H with a known minimal GHD and a modification δ.

A breakdown of the widths of the instances before modifications is shown in Table 5.1. The row marked “Original” shows how many original instances have minimal width k, for each $k \in [2, 10]$, while the “Updated” row shows how many updated hypergraphs stemmed from the original hypergraphs by performing the modifications in the way described above, for each width $k \in [2, 10]$. Finally, the column “Total” reports on the total number of instances in each category. We can see that the most hypergraphs have width $k \in [3, 7]$.

5.4.2 Updating GHDs

We compare our methods in the task of updating GHDs after a hypergraph modification happens using the dataset described above. To reduce the effect of variance, we only report on the 26013 instances for which it took *ClassicGo* more than 15 milliseconds to compute a decomposition. In the “easier” cases, it is reasonable to just use *ClassicGo* instead of the more sophisticated *Update*. If we move the threshold to any $t > 15$, the superiority of *Update* becomes even clearer. This suggests that our approach is even more fruitful when applied to “hard” cases.

Since our *Update* approach is built on top of *ClassicGo*, we will use only the latter as a baseline for our experiments. As it will be evident, this is also justified by the fact that *ClassicGo* performs better than *ClassicLeo* on average.

The results for each class of modification are shown in Table 5.2. The column *Positive* contains the percentage of cases where the width of the hypergraph did not increase due to the modification. The column *Better* contains the percentage of instances in which *Update* outperformed *ClassicGo*. In the next three columns we record the geometric means (in milliseconds) for *ClassicGo*, *Update*, and *ClassicLeo*. We then report on the speedup, which is defined as the ratio between *ClassicGo* and *Update* runtimes, via the geometric mean of all speedups. In the last columns, we compare the number of *exclusive timeouts* for each solver. For instance, the column *ClassicGo* reports on the number of instances that timed out for *ClassicGo*, but neither for *Update* nor *ClassicLeo*. Finally, for each operation, we show the number of instances that timed out for all methods. Since computing T^* takes far less than a millisecond for all of our instances, the time is not reported explicitly.

In order to compare the different approaches, we adopt the same methodology that was adopted in [52], i.e., we compare mean running times and number of instances that timed out. Overall, Table 5.2 clearly demonstrates the significant benefits of using *Update*. For every modification class, the *Update* mean time is significantly lower than the other approaches. The mean speedups are very high throughout all modifications even in the most difficult cases, i.e., DelEq and AddVar. We also see that *ClassicLeo* seems to have a hard time with most of the test instances, and has by far the most timeouts and the larger mean times in comparison with the other two methods.

Interestingly, *Update* seems to be particularly well suited for DelConstr and AddEq modifications. In theory, DelConstr is problematic since the deleted edge could have covered an arbitrarily complex structure. However, it seems that this occurs rarely in practice and deleting an edge simplifies the hypergraph instead. This is clearly apparent in the observation that 99.48% of DelConstr instances were positive, i.e., the width did not increase by deleting a constraint.

The *Better* column shows that *Update* is faster than *ClassicGo* in 78.8% of cases on average. This is despite the fact that many instances were solvable by *ClassicGo* in less than 40 milliseconds and *Update* has an additional overhead because of the scene mapping creation. Another source of slowdowns are negative instances ($ghw(\delta(H)) > k$), where the entire search space needs to be explored. In this case, the scene mapping is of little use and its creation only causes delays. Moreover, the *Timeout* columns show that *Update* solves $\approx 94\%$ of the instances, while *ClassicGo* and *ClassicLeo* solve 89.2% and 60.2% of them, respectively.

The *Positive* column shows that elementary modifications do not change the width of the hypergraph in 91.26% of cases. This is somewhat surprising as some of the modifications (e.g., AddConstr, AddEq) can lead to severe structural changes that intuitively increase how connected parts of the hypergraph are. Despite this, we observe that all modifications increase *ghw* only rarely. We believe that this is because the width of a hypergraph effectively captures only the most structurally complex part. With higher width and larger hypergraphs it becomes more common that large parts of the hypergraph are less complex than the width suggests. Even if a modification makes such a simpler part of the hypergraph more complex, our observations illustrate that this rarely affects the overall width.

Table 5.3: Statistics for *ClassicGo*, *Update*, and *ClassicLeo* shown by the width of the update instances. *Mean ClassicGo*, *Mean Update*, and *Mean ClassicLeo* are in milliseconds. All mean times were rounded to the closest integer, all other non-integer numbers were rounded to two decimal places.

Width	Count	Positive (%)	Better (%)	Mean ClassicGo	Mean Update	Mean ClassicLeo	Mean Speedup
2	732	27.87	34.29	906	784	59227	1.16
3	1582	66.81	70.29	773	191	220364	4.05
4	3511	97.09	80.03	762	24	29063	31.17
5	2241	100.00	88.26	866	13	9624	69.20
6	4055	100.00	82.22	1049	34	142452	30.71
7	1056	100.00	85.61	251	17	201694	14.86
8	28	100.00	60.71	794	148	773108	5.35
9	3	100.00	66.67	6336	54	1559313	118.41

Table 5.4: Statistics for *ClassicGo*, *Update*, and *ClassicLeo* shown by the width of the *positive* update instances. All mean times were rounded to the closest integer.

Width	Count	Better (%)	Mean ClassicGo	Mean Update	Mean ClassicLeo	Mean Speedup
2	204	55.39	54	26	103151	2.05
3	1057	83.73	122	15	176450	8.20
4	3409	81.93	609	18	30376	34.62
5	2241	88.26	866	13	9624	69.20
6	4055	82.22	1049	34	142452	30.71
7	1056	85.61	251	17	201694	14.86
8	28	60.71	794	148	773108	5.35
9	3	66.67	6336	54	1559313	118.41

Some evidence for these intuitions can be seen in Table 5.3 where we can observe a clear trend that the width increase is less common when the width gets higher (which itself strongly correlates with hypergraph size, see [52]). This leads to higher *Positive %*. Note that the problem is not trivial even though the positive rate approaches 100% for the most complex hypergraphs. Even if the width stays the same, a minimal width GHD for the new hypergraph may be entirely different than the input GHD (cf., the *Better %* column). As we ultimately need a GHD to algorithmically exploit low width, the high rate of positive instances therefore does not simplify the problem. Moreover, these percentages are obtained by running our experiments on a vast number of hypergraphs, thus showing that our formulation of the problem is realistic and relevant in practice as it is supported by the data. Table 5.4 and Table 5.5 give additional insights into positive and negative update instances, respectively.

Table 5.5: Statistics for *ClassicGo*, *Update*, and *ClassicLeo* shown by the width of the *negative* update instances. All mean times were rounded to the closest integer.

Width	Count	Better (%)	Mean ClassicGo	Mean Update	Mean ClassicLeo	Mean Speedup
2	528	26.14	2695	2910	47800	0.93
3	525	43.24	31951	32648	344718	0.98
4	102	16.67	1337066	1434802	6633	0.93

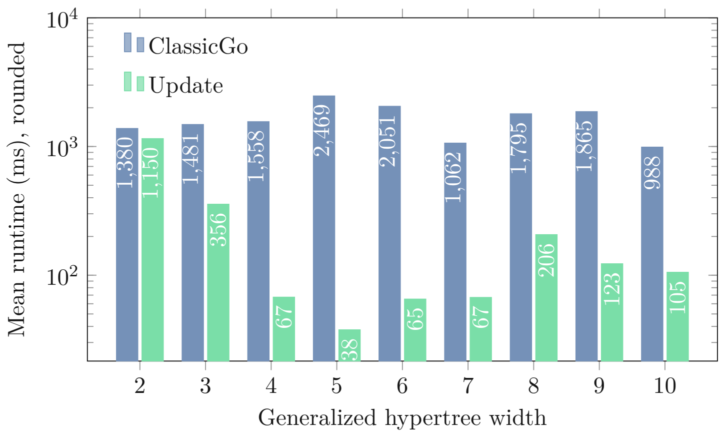
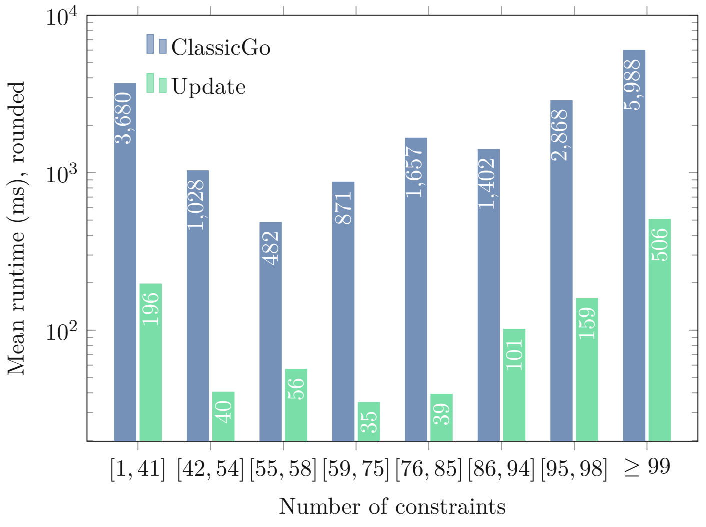


Figure 5.5: Geometric mean runtimes (log. scale) of *ClassicGo* and *Update* w.r.t. *ghw*.

We also investigated how our approach behaves with increasing *ghw* of the input decomposition as well as in relation to hypergraph size (in number of constraints and vertices, separately). The results of both studies are summarized in Figure 5.5 and Figure 5.6. In creating these plots, we partitioned the x-axis into intervals of similar size. Note that the runtimes are given on a logarithmic scale. Since *ClassicLeo* is more than one order of magnitude slower than the other two methods, we do not report on it. We see that beginning from width 3, *Update* provides significantly better mean runtimes than *ClassicGo*, and the speedup generally increases as well.

We observe that the superiority of *Update* becomes more pronounced as the input hypergraphs become larger. Intuitively, this is explained by the fact that the modification usually affects a smaller fraction of the hypergraph as the size increases. Hence, if it is possible to replace the δ-mutable subtree and reuse much of the old decomposition, as shown in Section 5.3, then the strengths of *Update* are emphasized. In practice, this is particularly promising since the recomputation of a GHD is problematic especially for larger instances.



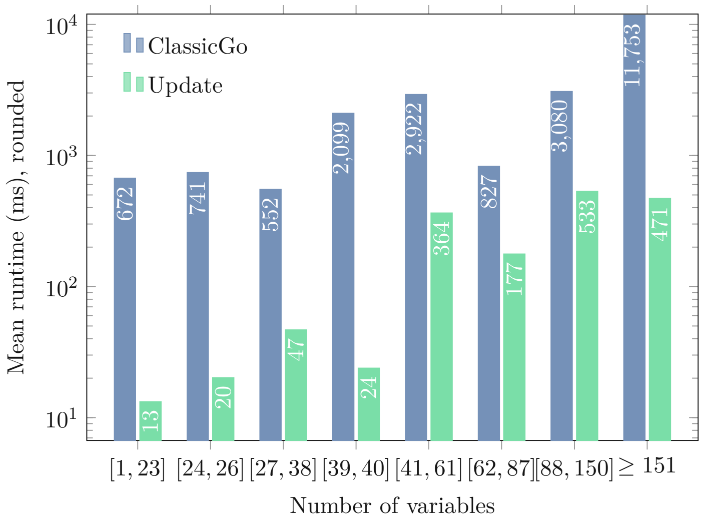


Figure 5.6: Geometric mean runtimes (log. scale) of *ClassicGo* and *Update* w.r.t. instance size.

5.5 Proof of Theorem 5.1

The argument will require details of the reduction of 3-Sat to checking whether a hypergraph has *ghw* at most 2 by Gottlob et al. [69]. The reduction is highly technical and we recall the construction and key facts here for convenience. For full details we refer to [69]. It will be convenient to use $[n]$ for integer n to refer to the set $\{1, 2, \ldots, n\}$.

5.5.1 Reducing 3-Sat to Checking $ghw \leq 2$

We want to construct a hypergraph H consisting of three main parts: two versions of a gadget introduced below and a subhypergraph encoding the clauses of the 3-Sat instance. We first fix some notation. We write $[n]$ for the set $\{1, \ldots, n\}$. Extending this common notation, we write $[n; m]$ for the set of pairs $[n] \times [m]$. Furthermore, we refer to the element $(1, 1)$ of any set $[n; m]$ as min and to (n, m) as max.

For two disjoint sets M_1, M_2 and $M = M_1 \cup M_2$ the construction makes use of a gadget with vertices $V = \{a_1, a_2, b_1, b_2, c_1, c_2, d_1, d_2\} \cup M$ and edges $E_A \cup E_B \cup E_C$ as follows:

$$\begin{aligned}
E_A &= \{\{a_1, b_1\} \cup M_1, \{a_2, b_2\} \cup M_2, \{a_1, b_2\}, \{a_2, b_1\}, \{a_1, a_2\}\} \\
E_B &= \{\{b_1, c_1\} \cup M_1, \{b_2, c_2\} \cup M_2, \{b_1, c_2\}, \{b_2, c_1\}, \{b_1, b_2\}, \{c_1, c_2\}\} \\
E_C &= \{\{c_1, d_1\} \cup M_1, \{c_2, d_2\} \cup M_2, \{c_1, d_2\}, \{c_2, d_1\}, \{d_1, d_2\}\}
\end{aligned}$$

Let $\varphi = \bigwedge_{j=1}^{m} (L_j^1 \vee L_j^2 \vee L_j^3)$ be an arbitrary instance of 3-Sat with m clauses and variables $x_1, \ldots, x_n$. In addition to the vertices for two of the aforementioned gadgets, the reduction uses the following sets to construct the target hypergraph H:

Y, Y', Y_ℓ, Y'_ℓ: The sets $Y = \{y_1, \ldots, y_n\}$ and $Y' = \{y'_1, \ldots, y'_n\}$ will encode the truth values of the variables of φ. Y_ℓ (Y'_ℓ) are the sets $Y \setminus \{y_\ell\}$ $(Y' \setminus \{y'_\ell\})$.

A, A', A_p, A'_p: We have sets $A = \{a_p \mid p \in [2n+3; m]\}$ and $A' = \{a'_p \mid p \in [2n+3; m]\}$ with the following important subsets:

$$\begin{aligned}
A_p &= \{a_{\min}, \ldots, a_p\} & \qquad \overline{A_p} &= \{a_p, \ldots, a_{\max}\} \\
A'_p &= \{a'_{\min}, \ldots, a'_p\} & \qquad \overline{A'_p} &= \{a'_p, \ldots, a'_{\max}\}
\end{aligned}$$

S: First define $Q = [2n+3; m] \cup \{(0,1), (0,0), (1,0)\}$. Then, S is defined as $Q \times \{1,2,3\}$. The elements in S are pairs, which we denote as $(q \mid k)$. The values $q \in Q$ are themselves pairs of integers (i, j).

S_p: For $p \in [2n+m; m]$ we write S_p for the set $\{(p,1), (p,2), (p,3)\}$. And S_p^k for the singleton $\{(p \mid k)\}$ for $k \in \{1,2,3\}$.

The vertices of H are as follows.

$$\begin{aligned}
V(H) = S \;\cup\; A \;\cup\; A' \;\cup\; Y \;\cup\; Y' \;\cup\; \{z_1, z_2\} \cup \\
\{a_1, a_2, b_1, b_2, c_1, c_2, d_1, d_2, a'_1, a'_2, b'_1, b'_2, c'_1, c'_2, d'_1, d'_2\}.
\end{aligned}$$

The edges of H are defined below. First, we take two copies of the gadget H_0 described above:

- Let $H_0 = (V_0, E_0)$ be the hypergraph of the lemma described at the beginning of the section with $V_0 = \{a_1, a_2, b_1, b_2, c_1, c_2, d_1, d_2\} \cup M_1 \cup M_2$ and $E_0 = E_A \cup E_B \cup E_C$, where we set $M_1 = S \setminus S_{(0,1)} \cup \{z_1\}$ and $M_2 = Y \cup S_{(0,1)} \cup \{z_2\}$.
- Let $H'_0 = (V'_0, E'_0)$ be the corresponding hypergraph, with $V'_0 = \{a'_1, a'_2, b'_1, b'_2, c'_1, c'_2, d'_1, d'_2\} \cup M'_1 \cup M'_2$ and E'_A, E'_B, E'_C are the primed versions of the edge sets $M'_1 = S \setminus S_{(1,0)} \cup \{z_1\}$ and $M'_2 = Y' \cup S_{(1,0)} \cup \{z_2\}$.

Beyond the gadget, H contains the following edges.

- $e_p = A'_p \cup \overline{A_p}$, for $p \in [2n+3; m]^-$,
- $e_{y_i} = \{y_i, y'_i\}$, for $1 \leq i \leq n$,
- For $p = (i, j) \in [2n+3; m]^-$ and $k \in \{1, 2, 3\}$:

$$e_p^{k,0} = \begin{cases} \overline{A_p} \cup (S \setminus S_p^k) \cup Y \cup \{z_1\} & \text{if } L_j^k = x_\ell \\ \overline{A_p} \cup (S \setminus S_p^k) \cup Y_\ell \cup \{z_1\} & \text{if } L_j^k = \neg x_\ell, \end{cases}$$

$$e_p^{k,1} = \begin{cases} A'_p \cup S_p^k \cup Y'_\ell \cup \{z_2\} & \text{if } L_j^k = x_\ell \\ A'_p \cup S_p^k \cup Y' \cup \{z_2\} & \text{if } L_j^k = \neg x_\ell. \end{cases}$$

- $e^0_{(0,0)} = \{a_1\} \cup A \cup S \setminus S_{(0,0)} \cup Y \cup \{z_1\}$
- $e^1_{(0,0)} = S_{(0,0)} \cup Y' \cup \{z_2\}$
- $e^0_{\max} = S \setminus S_{\max} \cup Y \cup \{z_1\}$
- $e^1_{\max} = \{a'_1\} \cup A' \cup S_{\max} \cup Y' \cup \{z_2\}$

The key GHD. The hypergraph construction above is such that only certain (if any) width 2 GHDs are possible for H_φ. In particular, in [69] it is shown extensively that any width 2 GHD of H_φ needs to be a line, i.e., it has no branching. Furthermore, the gadget construction is used to specify two blocks of nodes that need to be at the two ends of the line, indirectly fixing the possible nodes between them. Here it is enough to consider the standard GHD that can be constructed when a satisfying assignment σ for φ is known. Recall, $x_1, \ldots, x_n$ are the variables of φ and let

$$Z = \{y_i \in Y \mid \sigma(x_i) = 1\} \cup \{y'_i \in Y' \mid \sigma(x_i) = 0\}.$$

The following line graph, together with B_u and λ_u labels from Table 5.6, describes a width 2 GHD $\mathcal{D}_\varphi$ for φ.

$$u_C - u_B - u_A - u_{\min \ominus 1} - u_{(1,1)} - \cdots - u_{(2n+3, m-1)} - u_{\max} - u'_A - u'_B - u'_C$$

In the following arguments we will make use of this basic structure to argue the existence of width 2 GHDs for other cases.

Table 5.6: Definition of B_u and λ_u for GHD of H.

$u \in T$	B_u	λ_u
u_C	$\{d_1, d_2, c_1, c_2\} \cup Y \cup S \cup \{z_1, z_2\}$	$\{c_1, d_1\} \cup M_1,\ \{c_2, d_2\} \cup M_2$
u_B	$\{c_1, c_2, b_1, b_2\} \cup Y \cup S \cup \{z_1, z_2\}$	$\{b_1, c_1\} \cup M_1,\ \{b_2, c_2\} \cup M_2$
u_A	$\{b_1, b_2, a_1, a_2\} \cup Y \cup S \cup \{z_1, z_2\}$	$\{a_1, b_1\} \cup M_1,\ \{a_2, b_2\} \cup M_2$
$u_{\min \ominus 1}$	$\{a_1\} \cup A \cup Y \cup S \cup Z \cup \{z_1, z_2\}$	$e^0_{(0,0)}, e^1_{(0,0)}$
$u_{p \in [2n+3;m]^-}$	$A'_p \cup \overline{A_p} \cup S \cup Z \cup \{z_1, z_2\}$	$e_p^{k_p,0}, e_p^{k_p,1}$
$u_{\max}$	$\{a'_1\} \cup A' \cup Y' \cup S \cup Z \cup \{z_1, z_2\}$	$e^0_{\max}, e^1_{\max}$
u'_A	$\{a'_1, a'_2, b'_1, b'_2\} \cup Y' \cup S \cup \{z_1, z_2\}$	$\{a'_1, b'_1\} \cup M'_1,\ \{a'_2, b'_2\} \cup M'_2$
u'_B	$\{b'_1, b'_2, c'_1, c'_2\} \cup Y' \cup S \cup \{z_1, z_2\}$	$\{b'_1, c'_1\} \cup M'_1,\ \{b'_2, c'_2\} \cup M'_2$
u'_C	$\{c'_1, c'_2, d'_1, d'_2\} \cup Y' \cup S \cup \{z_1, z_2\}$	$\{c'_1, d'_1\} \cup M'_1,\ \{c'_2, d'_2\} \cup M'_2$

5.5.2 Adapting the Argument to Updates

We now show how to use the construction for the reduction from 3-Sat to checking $ghw \leq 2$ to the update problem restricted to the stated classes of atomic updates. Recall that for an instance φ of 3-Sat, the constructed hypergraph H_φ has $ghw(H_\varphi) = 2$ if φ is satisfiable, and width 3 otherwise. Our plan is to manipulate H_φ in such a way that we can efficiently construct a width 2 GHD of $H' = \delta^{-1}(H_\varphi)$ (and H' is not acyclic) for δ in the respective classes. If such a modification $\delta \in \Delta$ always exists, then the satisfiability of φ many-one reduces to the decision version of SearchUpdateGHD(Δ) for inputs H', δ, and the width 2 GHD of H'. We are able to give such a reduction for the DelConstr and DelEq case. We will describe below how to handle the other operations using a slightly more involved strategy.

DelConstr. Here our goal is easy to reach. To obtain H' it is sufficient to add a large edge $e^* = V(H) \setminus \{d_1\}$ to H_φ is sufficient. Since d_1 has at least two distinct edges to other vertices (which are in e^*) we see that the resulting H' is not acyclic. Clearly then $ghw(H') = 2$ and it is trivial to construct an appropriate width 2 GHD.

DelEq. Let H' be the hypergraph performing a AddEq modification on vertices z_2 and a'_1 in H_φ, (using a'_1 to represent the vertex after the join). In particular this will merge edges containing M'_1, M_2 and M'_2 in the two gadgets as well as all edges of form $e_p^{k,1}$ (as well as some linking edges in the gadget). The resulting edge e^* is of the form

$$e^* = S \cup Y \cup Y' \cup A' \cup \{a'_1, a'_2, b'_1, b'_2, c'_2, d'_2, b_2, c_2, d_2, z_1\}$$

The GHD $\mathcal{D}_\varphi$ given above can then be adapted in the following manner to yield a GHD of width 2 for H'. Replace all edges that contained z_2 or a'_1 in covers λ_u by the new e^*. Note that this affects all nodes in the GHD. Then add e^* to the bag of every node. Clearly, all edges of H_φ are still covered and our new edge e^* is covered in every node. Finally, H' is not acyclic as some cycles in the gadgets remain untouched by the merge. For example, the edges $\{c_1, d_2\}, \{c_1, c_2\}$ form an α-cycle with e^* (note that e^* does not contain c_1).

5.5.3 The Complex Cases – AddVar, AddConstr, and AddEq

For the other modification classes we will now slightly change our strategy and instead show how to decide satisfiability of 3-Sat via a polynomial number of calls to SearchUpdateGHD. Note that we use the returned new decompositions from the calls and thus can not directly derive $\mathcal{NP}$-hardness by Turing reduction of the SearchUpdateGHD decision problem. Formally, for a class $\mathcal{C}$ of updates, instead of a single H' we construct a sequence $H'_0, \dots, H'_\ell$ with $H'_\ell = H_\varphi$, ℓ polynomial in the size of φ, and for each $i \in [\ell]$ there is a $\delta_i \in \mathcal{C}$ such that $\delta_i(H'_{i-1}) = H'_i$. We will show that for all $0 \leq i \leq \ell$, $ghw(H_i) \leq 2$ if and only if φ is satisfiable.

Suppose now that we can construct such H'_0, sequence of modifications $\delta_1, \delta_2, \dots, \delta_\ell$, as well as a width 2 GHD for H'_0 efficiently. If SearchUpdateGHD($\mathcal{C}$) were feasible in polynomial time, then we could verify $ghw(H_\varphi) \leq 2$ in polynomial time by iteratively constructing a GHD for it from successive calls to SearchUpdateGHD, starting from the known GHD of H'_0. As stated previously, φ is satisfiable if and only if $ghw(H_\varphi) \leq 2$ and thus this would yield a polynomial procedure for solving 3-Sat.

AddVar. The desired sequence of hypergraphs and modifications is defined via δ_i^{-1} being the modification that removes vertex y'_i from H'_i. Thus, $\ell = n$ and all $\delta_i \in$ AddVar. Observe that AddVar modifications can never decrease ghw, that is $ghw(H'_{i-1}) \leq ghw(H'_i)$ for all $i \in [\ell]$. This can be easily observed since their inverse, the modifications of DelVar, produce an induced subhypergraph and thus can not increase ghw. Thus, we see that if $ghw(H'_0) = 2$ and $ghw(H'_\ell) = 2$ (which is equivalent to φ being satisfiable), then $ghw(H'_\ell) = 2$ for all $1 \leq i \leq \ell$. We therefore see that this gives us a sequence of modifications as described above.

Recall that in the original GHD $\mathcal{D}_\varphi$ given above, the set Z is derived from a satisfying assignment for φ. For H'_0 we can simply set $Z = \emptyset$ (and remove all y'_i from the bags) to obtain a width 2 GHD. Only the edges e_{y_i} relied on Z to be covered in H_φ, but in H'_0 they are all singletons $\{y_i\}$ and thus always covered in the first gadget. Hence, we can construct a width 2 GHD for H'_0 (which is cyclic) and as described above, a linear number of calls to SearchUpdateGHD(AddVar) are sufficient to decide whether φ is satisfiable.

AddConstr. We again argue via a sequence $H'_0, \dots, H'_\ell$ of hypergraphs and modifications $\delta_1, \dots, \delta_\ell \in$ AddConstr with $\delta_i(H'_{i-1}) = H'_i$. In contrast to the AddVar case, AddConstr modifications can decrease ghw and our use of such a sequence thus depends on particular properties of our choice of modification sequence.

We define our sequences via δ_i^{-1} being the modification (in DelConstr) that deletes edge e_{y_i} from H_φ. The construction of a width 2 GHD for H'_0 is the same as for $\mathcal{D}_\varphi$ above but with $Z = \emptyset$. The function of Z is only to connect $u_{\min \ominus 1}$ and $u_{\max}$ in a way such that every e_{y_i} is covered in either one of the respective bags. Since H'_0 no longer contains those edges, this is still satisfied with $Z = \emptyset$. It is not difficult to verify that Z was not used to cover any other edges in H_φ and therefore the correctness of the resulting GHD. Now, suppose that $ghw(H_\varphi) = 2$, then there exists some width 2 GHD $\mathcal{D}_\varphi$ of the form shown above. Note that no e_{y_i} edge is used in a λ_u set for this GHD but all of them are covered in some bag. In consequence, $\mathcal{D}_\varphi$ is also a GHD for every hypergraph H'_i in our sequence, meaning every hypergraph in the sequence has *ghw* 2 iff $ghw(H_\varphi) = 2$. Note that while it is hard to find $\mathcal{D}_\varphi$, the key point here is that a width 2 GHD for the special case H'_0 can always be found easily.

We can now proceed as in the AddVar case. Start from input H'_0, δ_1, and the width 2 GHD of H'_0 as described above and call SearchUpdateGHD(AddConstr) to find a width 2 GHD for H'_1. Iterating this process, we either arrive at some H'_i for which $ghw(H'_i) > 2$ and reject or we show that $ghw(H_\varphi) \leq 2$. In the former case, we have by the argument above that then also $ghw(H_\varphi) > 2$. Thus, we can correctly decide whether $ghw(H_\varphi) \leq 2$ – and therefore also 3-Sat – using linearly many calls of SearchUpdateGHD(AddConstr).

AddEq. We construct the initial hypergraph H'_0 from H_φ, by replacing every edge $e_{y_i} = \{y_i, y'_i\}$ by the edge $e^*_i = \{y_i, \star_i\}$. Consider the sequence $\delta_1, \ldots, \delta_n$ such that $\delta_i \in$ AddEq merges $\star_i$ into y'_i, i.e., $\star_i$ is replaced in every edge by y'_i. It is easy to see that $H'_n = H_\varphi$ and if $ghw(H'_i) = 2$ for all $i \in [n]$, then $ghw(H_\varphi) = 2$. We will first argue that H'_0 has *ghw* 2 and that witnessing GHD can be found easily. Then we show that if $ghw(H_\varphi) = 2$, then $ghw(H'_i) = 2$ for all $i \in [n]$. All together this again means that it is possible to decide 3-Sat using a linear number of calls to SearchUpdateGHD(AddEq).

The decomposition for H'_0 is again based on $\mathcal{D}_\varphi$ with $Z = \emptyset$. Observe that $\mathcal{D}_\varphi$ does not use any e_{y_i} as a cover and thus the only concern with adapting it for H'_0 is making sure that every e^*_i is covered in some bag. To that end, add nodes u^*_i, for $i \in [n]$ as children of $u_{\min \ominus 1}$ with $B_{u^*_i} = e^*_i$ and cover $\lambda_{u^*_i} = \{e^*_i\}$. The connectedness condition is clearly not violated by these new nodes and every e^*_i is now covered. Let $\mathcal{D}_0$ be the GHD described here and note it clearly has width 2 (and H'_0 is not acyclic).

To see that every H'_i for $1 \leq i \leq n$ has $ghw(H'_i) = 2$, if $ghw(H_\varphi) = 2$, we now proceed in similar fashion to the argument for AddConstr. Since we assume that $ghw(H_\varphi) = 2$, there exists a satisfying assignment σ for the variables of φ. Let $Z = \{y_i \mid \sigma(x_i) = 1\} \cup \{y'_i \mid \sigma(x_i) = 0\}$ be the set as in the original definition of $\mathcal{D}_\varphi$. Let $\mathcal{D}_i$ be the GHD obtained from $\mathcal{D}_\varphi$ using this Z and for all j s.t. $i < j \leq n$, add nodes u^*_j as children of $u_{\min \ominus 1}$ as in the construction of $\mathcal{D}_0$ above. By construction, H'_i contains the edges e_{y_j} for $j \leq i$ and edges e^*_j for $j > i$. It is then straightforward to verify that $\mathcal{D}_i$ indeed is a width 2 GHD for H'_i. Thus, as described above, we can use a linear number of calls to SearchUpdateGHD(AddEq) to decide 3-Sat. Consequently, if $\mathcal{P} \neq \mathcal{NP}$, then SearchUpdateGHD (AddEq) can not be solvable in polynomial time.

5.6 Summary

In this chapter, we dealt with the problem of updating a GHD when the original hypergraph for which it was computed changes. After describing classes of elementary hypergraph modifications, we defined the SearchUpdateGHD and studied its complexity. We proved that this problem is as hard as computing a new decomposition of the modified hypergraph anew for almost all of our modifications classes.

Despite this strong theoretical result, we proposed the notion of δ-mutable subtrees, intending to clearly define which portion of a GHD is affected by a hypergraph modification and must be recomputed. Based on this notion, we devised a framework for the practical recomputation of GHDs in the face of hypergraph modifications that reuses those parts of the GHD that are still valid for the modified hypergraph. This method mitigates the costs of a total recomputation and is easily implementable in existing top-down decomposition algorithms.

Finally, we extended one classical algorithm with our techniques to handle GHD updates and extensively compared it against two classical algorithms in recomputing a GHD after a hypergraph modification. Even though the problem is theoretically hard, the results of our study point out that our approach, on average, greatly speeds up the computation of GHDs in response to elementary modifications.

CHAPTER 6

Grounding Planning Problems with Decompositions

In previous chapters, we investigated the problem of computing GHDs from different angles. After developing a novel algorithm to compute GHDs efficiently for a large class of hypergraphs, which we called `BalSep`, we used it to study the structural properties of real-world CQs and CSPs. We discovered that we could compute GHDs efficiently for most of them. Furthermore, we extended `BalSep` to deal with a dynamic scenario where given a hypergraph H and a GHD of H, we want to update the input GHD in response to a modification to H. The resulting algorithm exploiting the old GHD is faster than computing a new GHD from scratch. Yet we did not use GHDs to solve the database and AI problems we are interested in.

The opportunity arises in the intersection between these two fields. Corrêa et al. established a close connection by using query optimization techniques to solve the successor generation problem in lifted planning problems [39]. Planning problems are a typical example of AI problems. Here, given an initial state of the world, an appropriate sequence of actions has to be selected to reach the desired world state. Typically, users describe a planning task using a first-order formalism. If the search for a solution is performed directly on this representation, we talk about *lifted planning problems*. The decision version of this problem is EXPSPACE-complete and thus extremely difficult [44]. In contrast, *grounded planning problems* are first grounded into a propositional representation. Asking whether a plan exists on a grounded representation is PSPACE-complete [32]. The successor generation problem refers to the computation of the world states reachable from the current state by applying a single action. Since an action A can be applied to the current state of the world s only if its precondition $pre(A)$ is satisfied in s, the challenge lies in performing this check efficiently. Corrêa et al. point out that checking if a precondition $pre(A)$ is satisfied in a state s and computing all possible action instantiations is equivalent to query answering. This fact is self-evident when we consider s as a database and $pre(A)$ as a query to answer on s.

On the other hand, most planners ground planning tasks before searching for a solution. While the grounding phase might produce an exponential blowup, this procedure is typically worth it because it makes the search exponentially easier. The state-of-the-art planners work on grounded representations [26, 88, 86, 92]. Nevertheless, this approach limits the practical applicability of planning to cases where the grounded representation is “small enough” [100] and planners struggle as planning tasks become larger and harder [84, 107]. The size of the grounding is just one problem. Grounding a planning task requires solving the successor generation problem multiple times. Since this is equivalent to query answering, a grounder incurs the problem of finding a join plan minimizing the size of intermediate results. Fortunately, it is necessary to ground a relaxed version of the task where all queries are CQs, thus making the problem easier [87]. Yet it does not disappear.

The problem of grounding first-order representations has been extensively studied for Answer Set Programming (ASP) [17]. Solving a problem with ASP means encoding it into a logic program whose answer sets correspond to solutions. Similarly to planners, the most popular ASP solvers ground the first-order representation of the problem into a set of propositional atoms before searching for a solution. In systems like *lparse/smodels* [111] or *gringo/clasp* [59], the grounder is strictly separated from the search component while the *DLV* system [102] integrates these two phases. Morak and Woltran proposed a decomposition method of ASP non-ground rules based on tree decompositions [110], which helped in some cases where too many variables are present in a single logic rule. Even though planning problems could benefit from this method, Morak and Woltran designed it for the much more complex case of ASP rules where negation and disjunction possibly appear. Calimeri et al. [33] improved this method for the *DLV* system. They split rules heuristically using tree decompositions, but only if this reduces the estimated cost of grounding. In this case, decompositions are not directly accessible as their computation is tightly intertwined with the resolution phase, making it complicated to analyze this method in a different context. Nevertheless, in theory, GHDs are more general than TDs and have lower widths. This fact translates into an increased potential of reducing intermediate results. Additionally, a more tailored method of grounding the logic programs corresponding to simplified planning tasks would be desirable.

The theme of this chapter is the grounding of planning problems using structural decomposition methods. We propose a novel method based on generalized hypertree decompositions to split non-ground rules of logic programs corresponding to relaxed planning tasks. These programs have the characteristic that the body of every rule is a conjunctive query. Therefore, GHDs perfectly cover this case. For every rule of the logic program, we generate the corresponding hypergraph and decompose it. Following the structure of the decomposition, we split the original rule into several smaller rules to avoid the explosion of intermediate results. Their size is indeed exponentially bounded by the width of the decomposition. Afterward, we measure the grounding performances of different logic program decompositions. We compare our approach against two other methods: the rule decomposition used by the state-of-the-art planner *Fast Downward* [86, 87], and the tree-decomposition-based split by Morak and Woltran [110]. Moreover, we use the results of grounding the original logic program as a baseline. This analysis shows that decomposing logic programs improves the grounding running time for planning tasks. While GHDs outperform other methods in a few notable cases, the tree-decomposition-based method performs better on average, even if GHDs are a generalization of TDs. We hypothesize that this is caused by the decomposition shape and that low width as a parameter is insufficient to determine a decomposition's quality. This fact highlights the necessity of investigating what "good" decompositions are, i.e., which parameters are crucial for this assessment.

The rest of the chapter is structured as follows. In Section 6.1, we outline classical planning problems giving intuitions about the complexity and the way planners solve them. Section 6.2 introduces the problem of grounding in planning problems. We pay particular attention to decomposition methods designed to avoid blowup of intermediate results during grounding. Additionally, we propose our novel algorithm based on GHDs. We describe our experiments in Section 6.3, where we empirically evaluate the decomposition methods presented in the previous section. Finally, Section 6.4 summarizes our findings.

The research presented in this chapter is partially based on the article [38], written in collaboration with Augusto B. Corrêa, Markus Hecher, Malte Helmert, Florian Pommerening, and Stefan Woltran. In particular, we repropose here the grounding experiments carried out with the tree decomposition method by Morak and Woltran [110] adapted to the planning setting. Additionally, we present a novel contribution consisting of the decomposition of Datalog programs guided by GHDs.

6.1 Classical Planning Problems

In this section, we use first-order languages defined over a *function-free logical vocabulary* over an infinite set of *variables* $\mathcal{V}$, a finite set of *constants* $\mathcal{C}$, and a set of *predicate symbols* $\mathcal{P}$. We recall that an *atom* $P(\mathbf{t})$ is composed of a predicate symbol $P \in \mathcal{P}$ and a z-tuple of terms $\mathbf{t}$ (variables or constants), where z is the arity of P. We often refer to P just as an atom. We use $vars(\mathbf{t})$ to denote the set of variables in $\mathbf{t}$. With a slight abuse of notation, we often treat $\mathbf{t}$ as a set, even though it is formally an order sequence of variables or constants. Finally, we say that $P(\mathbf{t})$ is a *ground atom* if $vars(\mathbf{t}) = \emptyset$.

In its general meaning, a *planning problem* consists in selecting a sequence of *actions* that, given an *initial state* of the world, leads to a desired world state called *goal*. Typically, the high-level description of the world is provided in a logical formalism such as first-order logic. The environment is represented as a set of states, where each state is a complete description of the world at a particular time. A state is a set of ground atoms. We deal with first-order classical planning problems defined as follows.

Definition 6.1 (Planning Task)**.** *A first-order* planning task *is a tuple* $\Pi = \langle \mathcal{P}, \mathcal{C}, \mathcal{A}, I, G \rangle$ *where:*

- $\mathcal{P}$ *is a set of* predicate *symbols.*
- $\mathcal{C}$ *is a set of* objects*, or constants.*
- $\mathcal{A}$ *is a set of* action schemas*, where each action schema* $A \in \mathcal{A}$ *consists of three sets of atoms: a precondition* $pre(A)$*, an add list* $add(A)$*, and a delete list* $del(A)$*.*
- I *is a state referred to as* initial state*.*
- G *is a set of ground atoms called* goal condition*.*

An agent can change the current state of the world s through an action obtained by instantiating an action schema $A \in \mathcal{A}$. The set of possible action schemas $\mathcal{A}$ is fixed and given as input. Using $vars(A)$ to denote the variables appearing in any of the atoms in $pre(A) \cup add(A) \cup del(A)$, we say that A is a *ground action* if $vars(A) = \emptyset$. Moreover, a task where all actions are ground is a *ground task*. Not all actions can be performed in a certain state of the world s. Indeed, a ground action A is applicable in s if $pre(A) \subseteq s$. The current state of the world evolves according to the following relationship. The application of a ground action A in a state s leads to the *successor state* $succ(s, A) = (s \setminus del(A)) \cup add(A)$. A sequence of ground actions $\pi = \langle A_1, \ldots, A_n \rangle$ is applicable in a state s_0 and has $succ(s_0, \pi) = s_n$ if there are states $s_1, \ldots, s_{n-1}$ where A_i is applicable in s_{i-1} and $succ(s_{i-1}, A_i) = s_i$ for all $i \leq n$. At this point, we can formally define a solution for a planning problem. Given a goal condition G, we call *goal states* all those states s such that $G \subseteq s$. A *plan* is a sequence of ground actions π applicable in I such that $succ(I, \pi)$ is a goal state.

Example 6.1. *The Blocks World consists of a table where some boxes with same shape but different colors are arranged in a random configuration. In this world, a robotic arm has to move the boxes from the initial configuration to a given one. Some simplifying rules hold: location on the table as well as location on a block do not matter. Moreover, at most one block may be below or on top of a block. The robotic arm can hold only one block at a time and only one block can be moved at a time. Therefore, only a "clear" block can be moved, i.e., one that has no other block on top.*

This scenario can be modeled using four actions:

- $pickup(x)$*, lift clear block* x *from the table.*
- $putdown(x)$*, place the held block* x *directly onto the table.*
- $unstack(x, y)$*, pick up clear block* x *from block* y*.*
- $stack(x, y)$*, place block* x *onto clear block* y*.*

Table 6.1: Action's preconditions and effects for the Blocks World (Example 6.1).

Action	Precondition	Add	Delete
$pickup(x)$	$clear(x)$ $on\_table(x)$ $arm\_empty()$	$holding(x)$	$clear(x)$ $on\_table(x)$ $arm\_empty()$
$putdown(x)$	$holding(x)$	$clear(x)$ $on\_table(x)$ $arm\_empty()$	$holding(x)$
$stack(x, y)$	$clear(y)$ $holding(x)$	$clear(x)$ $on(x, y)$ $arm\_empty()$	$clear(y)$ $holding(x)$
$unstack(x, y)$	$on(x, y)$ $clear(x)$ $arm\_empty()$	$holding(x)$ $clear(y)$	$on(x, y)$ $clear(x)$ $arm\_empty()$

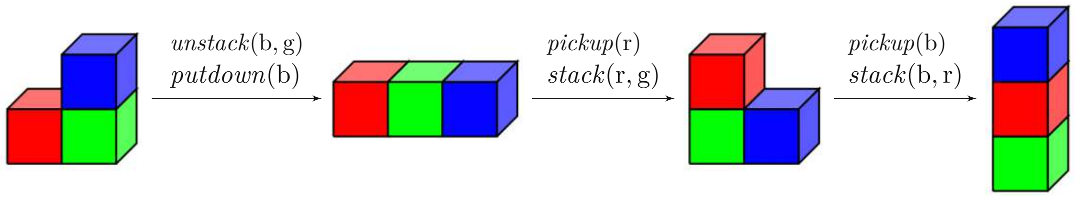
and five predicates:

- $on(x, y)$, *block* x *is on block* y.
- $on\_table(x)$, *block* x *is on the table.*
- $clear(x)$, *block* x *has nothing stacked on top.*
- $holding(x)$, *the arm holds block* x.
- $arm\_empty()$, *the arm holds no block.*

Table 6.1 shows the preconditions and effects for each action using the given predicates.

Let us use the following symbols for colors: blue (b), *green* (g), *and red* (r). *Suppose we are given the initial state shown in the leftmost part of Figure 6.1a, where* (r) *and* (g) *are on the table and* (b) *is on top of* (g). *We want to reach the goal state depicted in the rightmost part of Figure 6.1a, where a tower is formed by orderly piling up* (g), (r), *and* (b). *Then, the action sequence* $\langle unstack(\mathrm{b}, \mathrm{g}), putdown(\mathrm{b}), pickup(\mathrm{r}), stack(\mathrm{r}, \mathrm{g}), pickup(\mathrm{b}), stack(\mathrm{b}, \mathrm{r})\rangle$ *is a plan. Figure 6.1b shows the transition from the initial state to the goal. For ease of representation, we show the effects of two actions at a time.*

We can represent a planning task by its transition graph, i.e., a graph where vertices represent states of the world and edges represent the actions leading from one state to another. Then, a solution to a classical planning task is a path from the initial state I to a goal state $s_G \supseteq G$ in the transition graph. A naive way of finding a plan would be to perform *state-space search* by generating the transition graph and then using



(a) Illustration of a plan (adapted from [1]).

$$\begin{tikzcd}
\begin{array}{l} \mathit{arm\_empty}() \\ \mathit{on\_table}(\mathrm{r}) \\ \mathit{clear}(\mathrm{r}) \\ \mathit{on\_table}(\mathrm{g}) \\ \mathit{on}(\mathrm{b},\mathrm{g}) \\ \mathit{clear}(\mathrm{b}) \end{array} \arrow[r, "\begin{array}{l} \mathit{unstack}(\mathrm{b},\mathrm{g}) \\ \mathit{putdown}(\mathrm{b}) \end{array}"] & \begin{array}{l} \mathit{on\_table}(\mathrm{r}) \\ \mathit{clear}(\mathrm{r}) \\ \mathit{on\_table}(\mathrm{g}) \\ \mathit{clear}(\mathrm{g}) \\ \mathit{on\_table}(\mathrm{b}) \\ \mathit{clear}(\mathrm{b}) \\ \mathit{arm\_empty}() \end{array} \arrow[r, "\begin{array}{l} \mathit{pickup}(\mathrm{r}) \\ \mathit{stack}(\mathrm{r},\mathrm{g}) \end{array}"] & \begin{array}{l} \mathit{on\_table}(\mathrm{g}) \\ \mathit{on\_table}(\mathrm{b}) \\ \mathit{clear}(\mathrm{b}) \\ \mathit{clear}(\mathrm{r}) \\ \mathit{on}(\mathrm{r},\mathrm{g}) \\ \mathit{arm\_empty}() \end{array} \arrow[r, "\begin{array}{l} \mathit{pickup}(\mathrm{b}) \\ \mathit{stack}(\mathrm{b},\mathrm{r}) \end{array}"] & \begin{array}{l} \mathit{on\_table}(\mathrm{g}) \\ \mathit{on}(\mathrm{r},\mathrm{g}) \\ \mathit{on}(\mathrm{b},\mathrm{r}) \\ \mathit{clear}(\mathrm{b}) \\ \mathit{arm\_empty}() \end{array}
\end{tikzcd}$$

(b) State transitions during the execution of the plan.

Figure 6.1: A plan example in the Blocks World (Example 6.1).

any shortest-path algorithm to find a path from I to any s_G. While this algorithm is polynomial in the number of states, this number is exponential in the number of state atoms. Hence, it is exponential in the size of the input. Since constructing the transition graph is infeasible, planning algorithms avoid it by using concise descriptions of transition systems, such as first-order logic. Unfortunately, the problem does not get any easier. Given a lifted planning task Π, checking if there exists a plan for Π is an EXPSPACE-complete problem [44], while the same check problem for a grounded planning task is still PSPACE-complete problem [32].

Despite its hardness, several approaches to planning work well in practice. The most common is heuristic search planning which consists in searching the state space of the planning task with the help of a heuristic function h extracted from the planning task Π. A common approach consists in solving a simplified version of the original planning task known as *delete relaxation*. For grounded planning tasks, the delete-relaxed task is solvable in polynomial time [26, 88]. All actions' delete effects are removed in the delete-relaxed task. Solving such a task gives insightful information about the original task. Since no action removes any atom, the set of possible states of the relaxed task is an over-approximation of the set of possible states of the original task. If the goal is unreachable in the delete-relaxed task, it is also unreachable in the original task. Else, the heuristic guides the search for the goal as if no delete effects existed. This strategy proves to be quite successful in practice. Most state-of-the-art planners such as Fast Downward [86], FF [88], and LAMA [121] solve planning tasks by performing heuristic search on grounded representations.

6.2 Grounding Planning Tasks

Planning with grounded representations is exponentially easier than planning with lifted representations [44]. Grounding has an exponential cost because it consists of instantiating every action schema using every object. The hardness of planning is thus unavoidable. Nevertheless, more sophisticated algorithms can restrain the grounding size. Here we present several logic program decompositions designed to reduce the cost of grounding. After introducing the two decompositions by Helmert [87] and Morak and Woltran [110], we describe our grounding method based on generalized hypertree decompositions.

6.2.1 From Planning Tasks to Logic Programs

First-order planning tasks are often defined in a first-order modeling language called PDDL [85], which allows encoding complex features much more powerful than the ones in Definition 6.1. Perhaps the most used algorithm to transform a PDDL planning task into its grounded representation is the one proposed by Helmert in [87]. This four-step procedure executes the following tasks: normalization, invariant synthesis, grounding, and task generation. While each step could potentially slow the translation, Helmert reports that grounding typically consumes about 70% of the running time, thus being the most critical task.

A fundamental step in the translation is the definition of a Datalog program [134, 135] equivalent to the delete-relaxed version of the original planning task. Since this version does not use the full power of the PDDL formalism, we provide a simplified exposition of the translation based on our definition of first-order planning tasks.

Definition 6.2 (Datalog Programs [87])**.** *A* Datalog rule *is a first-order formula* $h(\mathbf{t}) \leftarrow p_1(\mathbf{t_1}), \ldots, p_m(\mathbf{t_m})$*, where* $p_i(\mathbf{t_i})$ *and* $h(\mathbf{t})$ *are atoms. The atom* $h(\mathbf{t})$ *is the rule* head*, while the conjunction of atoms* $p_1(\mathbf{t_1}), \ldots, p_m(\mathbf{t_m})$ *is the rule* body*. We assume that Datalog rules are universally quantified. Therefore, for a given Datalog rule* r *with free variables* $free(r) = \{v_1, \ldots, v_n\}$*, we define* $r_\forall = (\forall v_1, \ldots, v_n : r)$*. Similarly, for a set of Datalog rules* $\mathcal{R}$*, we define* $\mathcal{R}_\forall = \{r_\forall \mid r \in \mathcal{R}\}$*.*

A Datalog program *is a pair* $\langle \mathcal{F}, \mathcal{R} \rangle$*, where* $\mathcal{F}$ *is a set of ground atoms called* facts *and* $\mathcal{R}$ *is a set of Datalog rules called just* rules*.*

The canonical model *of a Datalog program* $\langle \mathcal{F}, \mathcal{R} \rangle$ *is the set of all ground atoms* φ *such that* $\mathcal{F} \cup \mathcal{R}_\forall \models \varphi$*.*

The Datalog program encodes the atom reachability problem for the planning task. Computing only "reachable" atoms avoids the unnecessary blowup of the naive algorithm. An atom is reachable if it is derivable by applying an appropriate action sequence from the initial state. Nevertheless, checking if a given atom is satisfied in any reachable state is as difficult as planning. Indeed, in general, grounding a Datalog program is intractable as the number of reachable atoms and actions might be exponential in the size of the program [137, 89, 40]. In practice, a conservative but tight approximation is computed.

Such an approximation must include all reachable facts and exclude as many unnecessary facts as possible. The delete-relaxed task [88] satisfies these conditions. Computing the set of reachable facts of the delete-relaxed task is easier than computing the set of reachable facts of the original task.

The delete-relaxed planning task assumes that inferred atoms will be true forever. This result follows from removing negative literals in action preconditions, effects, and goal conditions. Additionally, delete effects are ignored. Helmert efficiently computes the relaxed-task's set of reachable atoms with an algorithm called Datalog Exploration [87], which we will briefly describe in the next section.

Given a planning task $\Pi = \langle \mathcal{P}, \mathcal{C}, \mathcal{A}, I, G \rangle$, reachability in the delete-relaxed task Π^+ is represented as a Datalog program as follows. The idea is that a ground atom is reachable in Π^+ if and only if either it is true in the initial state or can be reached through an appropriate sequence of actions. Modeling this process as a Datalog program consists of the following steps. First, the set of all facts has to correspond to the initial state I. Then, rules are defined to model how actions allow the state to evolve. Finally, we need a rule checking that the goal of the relaxed task is reachable. Since Π^+ contains all the reachable atoms of Π, if Π^+ is unsolvable, then also Π must be unsolvable. In this case, we stop here since the search phase is unnecessary. Let us use p^+ to denote the *positive* literals in the rule's body. The Datalog program is constructed as follows.

- *Facts*: The set of facts $\mathcal{F} = I$ includes all ground atoms in the initial state.
- *Actions*: For each action schema $A \in \mathcal{A}$ with $vars(A) = \mathbf{t}$ and precondition $pre(A) = \{p_1^+(\mathbf{t_1}), \ldots, p_m^+(\mathbf{t_m})\}$, we generate the *action applicability rule*

  $$A\texttt{-applicable}(\mathbf{t}) \leftarrow p_1^+(\mathbf{t_1}), \ldots, p_m^+(\mathbf{t_m}).$$

  and for each atom $q(\mathbf{t'}) \in add(A)$ we generate the *effect rule*

  $$q(\mathbf{t'}) \leftarrow A\texttt{-applicable}(\mathbf{t}).$$

  where $\mathbf{t} = \bigcup_i \mathbf{t_i}$ and $\mathbf{t'} \subseteq \mathbf{t}$.
- *Goal*: For $G = \{p_1^+(\mathbf{t_1}), \ldots, p_m^+(\mathbf{t_m})\}$, we generate the *goal rule*

  $$goal() \leftarrow p_1^+(\mathbf{t_1}), \ldots, p_m^+(\mathbf{t_m})$$

  where each p_i^+ is ground.

The normalization step preceding the generation of the Datalog program guarantees that these rules are safe. The action applicability rules serve the additional purpose of avoiding the recomputation of common subexpressions. If absent, each effect rule body would repeat all action preconditions.

Example 6.2. *Recall the Blocks World described in Example 6.1. To obtained the relaxed version of the task described there, we need to translate the intial state and the goal condition into facts. Moreover, we need to remove all negative literals from actions' preconditions and effects. Since no negative literals appear in the preconditions, it is sufficient to ignore the delete effects. Then, the Datalog program of this task is:*

$$
\begin{aligned}
&\mathit{arm\_empty}().\\
&\mathit{on\_table}(\mathrm{r}).\\
&\mathit{clear}(\mathrm{r}).\\
&\mathit{on\_table}(\mathrm{g}).\\
&\mathit{on}(\mathrm{b}, \mathrm{g}).\\
&\mathit{clear}(\mathrm{b}).\\[1em]
&\mathit{pickup\text{-}applicable}(x) \leftarrow \mathit{clear}(x), \mathit{on\_table}(x), \mathit{arm\_empty}().\\
&\mathit{holding}(x) \leftarrow \mathit{pickup\text{-}applicable}(x).\\[1em]
&\mathit{putdown\text{-}applicable}(x) \leftarrow \mathit{holding}(x).\\
&\mathit{clear}(x) \leftarrow \mathit{putdown\text{-}applicable}(x).\\
&\mathit{on\_table}(x) \leftarrow \mathit{putdown\text{-}applicable}(x).\\
&\mathit{arm\_empty}() \leftarrow \mathit{putdown\text{-}applicable}(x).\\[1em]
&\mathit{stack\text{-}applicable}(x, y) \leftarrow \mathit{clear}(y), \mathit{holding}(x).\\
&\mathit{clear}(x) \leftarrow \mathit{stack\text{-}applicable}(x, y).\\
&\mathit{on}(x, y) \leftarrow \mathit{stack\text{-}applicable}(x, y).\\
&\mathit{arm\_empty}() \leftarrow \mathit{stack\text{-}applicable}(x, y).\\[1em]
&\mathit{unstack\text{-}applicable}(x, y) \leftarrow \mathit{on}(x, y), \mathit{clear}(x), \mathit{arm\_empty}().\\
&\mathit{holding}(x) \leftarrow \mathit{unstack\text{-}applicable}(x, y).\\
&\mathit{clear}(y) \leftarrow \mathit{unstack\text{-}applicable}(x, y).\\[1em]
&\mathit{goal}() \leftarrow \mathit{on\_table}(\mathrm{g}), \mathit{on}(\mathrm{r}, \mathrm{g}), \mathit{on}(\mathrm{b}, \mathrm{r}), \mathit{clear}(\mathrm{b}).
\end{aligned}
$$

6.2.2 Project-Join Decomposition

Helmert's grounding algorithm [87] consists of three steps: generation of the logic program, translation to normal form, and computation of the canonical model. The algorithm focuses on computing the canonical model of the Datalog program efficiently. To do so, Helmert defines a normal form based on rule decompositions that split rules into smaller ones with a maximum of two atoms in the body. The intuition behind this is that rules with fewer variables are expected to be easier to ground and produce smaller intermediate results. We refer to this decomposition as *Project-Join Decomposition.*

Definition 6.3 (Project-Join Decomposition [87]). *A first-order logic atom is* variable-unique *if it does not contain two occurrences of the same variable. A Datalog rule is* variable-unique *if its head and all atoms in its body are variable-unique.*

A Datalog rule is a projection rule *if it is variable-unique and of the form* $h(\mathbf{t}') \leftarrow p(\mathbf{t})$ *with* $vars(\mathbf{t}') \subseteq vars(\mathbf{t})$. *In other words, projection rules are unary rules where all variables in the head occur in the body.*

A Datalog rule is a join rule *if it is variable-unique and of the form* $h(\mathbf{t}) \leftarrow p_1(\mathbf{t_1}), p_2(\mathbf{t_2})$ *with* $vars(\mathbf{t_1}) \cup vars(\mathbf{t_2}) = vars(\mathbf{t}) \cup (vars(\mathbf{t_1}) \cap vars(\mathbf{t_2}))$. *In other words, join rules are binary rules where all variables in the head occur in the body. Additionally, all variables in the body but not in the head occur in both atoms.*

A Datalog program is in normal form *if all rules are projection or join rules.*

The names of these rules recall the projection and join operators of relational algebra and have similar semantics. Rules of the original Datalog program are decomposed by progressively applying two transformations.

- *Projection*: Given a rule $h(\mathbf{t}) \leftarrow p_1(\mathbf{t_1}), \ldots, p_i(\mathbf{t_i}), \ldots, p_m(\mathbf{t_m})$, pick an atom $p_i(\mathbf{t_i})$ with $i \in [1, m]$, such that there is a variable $v \in \mathbf{t_i}$ not occurring in any $\mathbf{t_j}$ with $j \neq i$ nor in $\mathbf{t}$. Then, substitute the original rule with the following two:

$$temp(\mathbf{t_i} \setminus v) \leftarrow p_i(\mathbf{t_i}).$$
$$h(\mathbf{t}) \leftarrow p_1(\mathbf{t_1}), \ldots, temp(\mathbf{t_i} \setminus v), \ldots, p_m(\mathbf{t_m}).$$

- *Join*: Given a rule $h(\mathbf{t}) \leftarrow p_1(\mathbf{t_1}), \ldots, p_i(\mathbf{t_i}), p_j(\mathbf{t_j}), \ldots, p_m(\mathbf{t_m})$, pick two atoms $p_i(\mathbf{t_i}), p_j(\mathbf{t_j})$ and substitute the original rule with the following two using $\mathbf{t}$ to denote all variables in $\mathbf{t_i} \cup \mathbf{t_j}$ occurring in any other $\mathbf{t_z}$ with $z \neq i, j$:

$$temp(\mathbf{t}) \leftarrow p_i(\mathbf{t_i}), p_j(\mathbf{t_j}).$$
$$h(\mathbf{t}) \leftarrow p_1(\mathbf{t_1}), \ldots, temp(\mathbf{t}), \ldots, p_m(\mathbf{t_m}).$$

A crucial point of the project-join decomposition is the selection of the atoms to project or join. The problem of choosing a good join order that minimizes the size of intermediate results is equivalent to the *query optimization problem* in a database setting. Therefore, computing a sequence of binary joins that minimizes the size of intermediate results is hard. Helmert uses a greedy algorithm to project away unnecessary variables as early as possible and to join atoms with many common variables first. This technique works well in many cases but fails in some important domains, as will be presented in the experimental section. Note that different decompositions lead to a diverse number of temporary predicates. This is one of the sources of the overhead of rule rewriting.

Starting from the normal form of the Datalog program, the algorithm proceeds by computing the canonical model of the program. This algorithm is reminiscent of the *seminaive evaluation* for computing the canonical model of a Datalog program [5]. In this phase, it is preferable to use an incremental approach that does not recompute sets of known facts. The algorithm maintains a set of derived facts, which is initially empty, and a queue of new facts, originally containing the atoms in I. At this point, a new fact is extracted from the queue and matched against all rules. Only preconditions containing that predicate symbol in the body will be evaluated. If a precondition is satisfied, a new fact is added to the queue unless it has been reached already. This loop terminates when the queue of new facts is empty. This means that a fix-point has been reached and no new derivations are possible. Finally, the grounded task is straightforwardly obtained from the canonical model.

6.2.3 Tree-decomposition-based Rules' Splitting

Helmert's algorithm does not consider the structure underlying the rules of the Datalog program. Morak and Woltran exploited the rule structure for grounding ASP programs [110], showing that logic programs of bounded treewidth can be grounded efficiently. Given a rule r, their method extracts the primal graph G_r from r, including the head of r. The vertex set $V(G_r)$ contains a vertex for each variable in r, while for each pair of vertices $v_1, v_2 \in V(G_r)$, $(v_1, v_2) \in E(G_r)$ if and only if the variables v_1, v_2 appear together in the same predicate in r. Then, it computes a tree decomposition of G_r used to rewrite the original logic program by introducing several temporary rules.

Given a rule r, a tree decomposition $\langle T, (B_n)_{n \in T} \rangle$, where $T = (N(T), E(T))$, of r is computed. Ideally, a decomposition of minimum treewidth is desirable because it corresponds to a rule with fewer variables. However, computing a minimum-treewidth decomposition is a hard problem. Then, approximate solutions are computed. Without loss of generality, the head predicate is assumed to be in the root of T. Subsequently, new rules are generated by visiting T in a bottom-up fashion. For each node $n \in N(T)$ with parent n', and thus except for the root, a new rule is created:

$$\mathit{temp}_n(\mathbf{Y}_n) \leftarrow \{p \in \mathit{body}(r) \mid \mathit{vars}(p) \subseteq B_n\} \cup \{\mathit{temp}_m(\mathbf{Y}_m) \mid (n, m) \in E(T)\}.$$

where $\mathbf{Y}_n = B_n \cup B_{n'}$ and $\mathbf{Y}_m = B_m \cup B_n$.

For the root node, a similar rule is generated, but the head $\mathit{temp}_{\text{root}}(\mathbf{Y}_{\text{root}})$ is substituted by $\mathit{head}(r)$. Additionally, all ground atoms in the body of the original rule are added here. This rule has the form:

$$\begin{aligned} \mathit{head}(r) \leftarrow & \{p \in \mathit{body}(r) \mid \mathit{vars}(p) \subseteq B_n\} \\ & \cup \{p \in \mathit{body}(r) \mid p \text{ is ground}\} \\ & \cup \{\mathit{temp}_m(\mathbf{Y}_m) \mid (\text{root}, m) \in E(T)\}. \end{aligned}$$

where $\mathbf{Y}_m = B_m \cap B_{\text{root}}$. These rules have the same instantiations of the original $\mathit{head}(r)$.

6.2.4 Generalized Hypertree Decompositions and Grounding

In contrast to tree decompositions, GHDs offer more guarantees on the size of intermediate results. Being $|r|$ the size of the biggest relation, the maximum size of intermediate results is $|r|^k$, where k is the *ghw* of the hypergraph. A Datalog program can be seen as a database. The facts can be grouped by predicate and become relations. Then, checking if a rule is triggered and computing all instantiations correspond to answering a query on a database. We are therefore interested in using GHDs for grounding.

Given a rule $r\colon h(\mathbf{t}) \leftarrow p_1(\mathbf{t_1}), \ldots, p_m(\mathbf{t_m})$, the hypergraph $H_r = (V(H_r), E(H_r))$ of r consists of $V(H_r) = \mathit{vars}(r)$ and $E(H_r) = \{h(\mathbf{t}), p_1(\mathbf{t_1}), \ldots, p_m(\mathbf{t_m})\}$, where we slightly abuse notation and use $h(\mathbf{t})$ to denote the set of variables $\mathbf{t}$. This hypergraph has an edge for each atom appearing in r, including its head. Then, given a GHD $\langle T, (B_u)_{u\in T}, (\lambda_u)_{u\in T}\rangle$ of H_r of width k, we split r by navigating the GHD in a bottom-up fashion. While we do not require that $h(\mathbf{t}) \in \lambda_{\text{root}}$ explicitly, it is important that all $\mathbf{t}$ variables are maintained during the bottom-up traversal of the GHD. We obtain this effect by preserving these variables in the head of the temporary rules generated along the path. First, for every edge cover λ_u, we create an additional rule

$$\mathit{cov}_u(\mathbf{t}') \leftarrow e_1, \ldots, e_k.$$

where $\mathbf{t}' = \{B_u \cup (B(\lambda_u) \cap \mathbf{t})\}$, and $e_1, \ldots, e_k \in \lambda_u$.

Then, the GHD is traversed. Keep in mind that the definition of GHD requires that for each $e \in E(H_r)$ there exists $u \in T$ such that $e \subseteq B(\lambda_u)$. However, it is not necessary for e to explicitly appear in any λ_u as long as it is covered by some λ_u. To enforce early projection and reduce the overall number of temporary rules generated by our algorithm, every time we visit a node u, we add all those atoms $e \in E(H_r)$ such that $\forall n \in T\colon e \notin \lambda_n$ yet $e \subseteq B(\lambda_u)$ but e has not been already added to any other $n \neq u$. We add these predicates as soon as possible. It is also important that, given a node u, all the head predicates of the rules generated by the children of u appear in the body of r_u. This is necessary to keep the values of the variables until the top. Then, the general rule is

$$\begin{aligned} h_u(\mathbf{t}'') \leftarrow & \mathit{cov}_u(\mathbf{t}') \\ & \cup \{e \in E(H_r) \mid e \subseteq B(\lambda_u) \wedge e \text{ is uncovered}\} \\ & \cup \{h_w(\mathbf{t_w}) \mid (u,w) \in E(T)\}. \end{aligned}$$

where $\mathbf{t}' = \{B_u \cup (B(\lambda_u) \cap \mathbf{t})\}$ and $\mathbf{t}'' = \{\mathbf{t}' \cup (\bigcup_{(u,w)\in E(T)} \mathbf{t_w} \cap \mathbf{t})\}$.

6.3 Experiments

In this section, we test our hypothesis that using structural decomposition methods, particularly GHDs, to ground Datalog programs leads to faster runtimes and more grounded instances for planning problems. After describing our experimental setting, we report on our experiments and discuss our findings.

6.3.1 Methodology

We want to know how the form of a Datalog program affects the grounding time and the number of grounded instances. We thus compare three different Datalog program rewritings using the relaxed planning task as a starting point. We recapitulate here the four Datalog variants described in the previous section:

- *original*: it refers to the Datalog program generated as described in Section 6.2.1.
- *fd*: it refers to Helmert's Project-Join Decomposition [87] described in Section 6.2.2.
- *lpopt*: it refers to Morak and Woltran's tree-decomposition-based rewriting [110] described in Section 6.2.3.
- *htd*: it refers to our rewriting method based on GHDs described in Section 6.2.4.

Except for *original*, all the decomposition methods introduce temporary predicates that do not appear in *original* and are unnecessary to solve the related planning tasks. While they could potentially lead to higher running times due to the evaluation of unnecessary rules, they also guide the grounder, which should benefit from them. These methods thus differ in the number of temporary predicates and the rules decomposition.

We generate both *original* and *fd* using the Fast Downward planner [86] available at `https://github.com/aibasel/downward`. This planner implements Helmert's grounding algorithm [87]. We recall that *fd* is a Datalog program consisting of "projection rules" projecting away unnecessary variables as soon as possible and "join rules" defining sequences of binary joins. Fast Downward's grounder computes these sequences greedily by joining atoms sharing the most variables. For each rule of *original*, we generate the hypergraph corresponding to the conjunction of atoms in the body of the rule. Recall that no negative atom appears in *original* since it models the relaxed planning task. We then decompose this hypergraph with different methods. To generate the *lpopt* Datalog program, we use tree decompositions computed by `lpopt` [23], which is available at `https://dbai.tuwien.ac.at/proj/lpopt`. This software uses heuristic methods to compute low-width tree decompositions. Nevertheless, the output comes without any guarantee of optimality: a tree decomposition of lower width might still exist. Finally, our *htd* Datalog program is based on GHDs generated by `BalancedGo` [75], which implements a parallel version of the `BalSep` algorithm presented in Chapter 3. In this case, we enforce that the computed GHDs have the lowest width. Our code is available at `https://github.com/dmlongo/decomp-grounding-planning`.

We use `gringo` [58] to ground our various Datalog programs. This program is a grounder for ASP problems using several optimization methods, such as on-the-fly join ordering based on selectivity estimates. This feature gives us some flexibility in the definition of our programs: we do not need to worry about the order of atoms in the rules. Only the splitting of the rules matters. On the contrary, the grounder used in Fast Downward was optimized for the specific *fd* decompositions giving it an unfair advantage over the other representations. Therefore, we exclude it from the comparison. On the other hand, the generality of `gringo` penalizes our methods because this grounder has been designed for a more general class of problems. In any case, all methods are penalized in the same way.

Our experiments use the hard-to-ground (HTG) data set by Lauer et al. [100] containing 862 tasks, divided into 8 different domains. This dataset is available at `https://github.com/abcorrea/htg-domains`. All experiments were run on Intel Xeon Silver 4114 processors running at 2.2 GHz. We use a time limit of 30 minutes and a memory limit of 16 GiB per task.

6.3.2 Results and Discussion

In our first experiment, we measured the treewidth and generalized hypertree width of the *original* Datalog program rules for each hard-to-ground planning domain [100]. We present the results in Table 6.2 aggregated per domain. Column A indicates whether action predicates are present in the Datalog program (✓) or not (✗). Except for the case of *genome-edit-distance*, these predicates involve all variables appearing in a rule, so their presence or absence influences both *tw* and *ghw*. We will provide more detail about this column later. Column N denotes the number of rules in a given domain and ✓/✗ variant. The following columns contain statistical information about *tw* and *ghw*, respectively. A "range" value of min–max shows that $w \in [\min, \max]$ where w is the width while an "average" value of $M \pm \sigma$ denotes that M is the standard mean and σ is the standard deviation of the width values.

Overall, Table 6.2 shows that treewidth is low for most domains except for *organic-synthesis*, *pipesworld-tankage*, and *childsnack*. We emphasize that the widths pf the tree decompositions reported here are not necessarily minimal as tree decompositions of lower width might exist. Indeed, the program `lpopt` computes tree decompositions heuristically without any minimality guarantee on the width. However, it is reasonable to assume that for low widths, the computed tree decompositions are close to having minimal width. On the other hand, we computed exact *ghw*. Hence, we conclude that almost all rules are acyclic. When they are not, the width is exactly 2. It is worth noting that at least one rule in the *genome-edit-distance* domain has a width equal to 0. This fact means that one rule is fully propositional, i.e., there are no variables to ground and no hypergraph to decompose. While we will return to this issue later, let us anticipate that removing action predicates has opposite effects on the width measures. The absence of these predicates generally leads to an increase in *ghw*. On the contrary, *tw* decreases, sometimes even drastically, as in the case of *organic-synthesis*. For the next experiment, we focused on the case described in Section 6.2 where action predicates are in the *original* Datalog program. In conclusion, since *ghw* is much lower than *tw* in all domains, we expect our method to perform better than others.

We proceeded with a direct comparison of *original*, *fd*, *lpopt*, and *htd* in the number of grounded instances and grounding time. For this experiment, we used the grounder `gringo` to ground the four different Datalog rewritings. We refer to the combination

Table 6.2: Widths (*tw* and *ghw*) computed for the original Datalog rules. Column *A* shows if action predicates are present. Column *N* shows the number of rules.

Domain	*A*	*N*	*tw*		*ghw*	
			range	average	range	average
blocksworld	✓	13	1–2	1.54±0.52	1–1	1±0.00
	✗	9	1–2	1.33±0.50	1–1	1±0.00
childsnack	✓	9	2–10	4.89±2.52	1–1	1.0±0.00
	✗	5	2–6	2.8±1.30	1–2	1.2±0.45
genome-edit-distance	✓	48	0–5	2.21±0.60	0–2	1.02±0.17
	✗	35	1–5	1.9±0.48	1–2	1.04±0.14
logistics	✓	12	3–4	3.17±0.39	1–1	1±0.00
	✗	6	2–3	2.83±0.41	1–2	1.83±0.41
organic-synthesis	✓	134	3–22	10.55±3.99	1–1	1.0±0.00
	✗	117	2–3	2.1±0.29	1–2	1.1±0.29
pipesworld-tankage	✓	26	9–12	10.62±1.53	1–1	1±0.00
	✗	22	3–4	3.64±0.49	2–2	2±0.00
rovers	✓	26	2–6	4.23±1.27	1–1	1±0.00
	✗	17	2–3	2.35±0.49	1–2	1.12±0.33
visitall-multidimensional	✓	12	4–6	5.17±0.00	1–1	1.0±0.00
	✗	8	4–6	5.17±0.00	2–2	2.0±0.00

Table 6.3: Number of grounded instances per domain for different Datalog rewritings. Timeout was set to 30 minutes. We abbreviate `gringo` with `G`, `lpopt` with `L`, and `htd` with `H`. Next to each domain we write the number of planning tasks. We separate the results into two blocks according to the presence or absence of action predicates. We mark in bold the best result for each block and domain.

Domain	**Action Predicates**				**No Action Predicates**			
	`G`	`G+FD`	`G+L`	`G+H`	`G`	`G+FD`	`G+L`	`G+H`
blocksworld (40)	**40**	**40**	**40**	**40**	**40**	**40**	**40**	**40**
childsnack (144)	**130**	**130**	**130**	**130**	**144**	**144**	**144**	**144**
genome-edit-dist. (312)	**312**	**312**	**312**	**312**	**312**	**312**	**312**	**312**
logistics (40)	**40**	**40**	**40**	**40**	**40**	**40**	**40**	**40**
organic-synthesis (56)	21	21	**22**	21	41	55	**56**	54
pipesworld-tankage (50)	**42**	**42**	**42**	**42**	**50**	**50**	**50**	**50**
rovers (40)	**40**	22	**40**	**40**	**40**	**40**	**40**	**40**
visitall-multidim. (180)	**174**	168	**174**	168	**180**	168	**180**	168
Total (862)	799	775	**800**	793	847	849	**862**	848

of `gringo` with the rewritings as `gringo+`R, where R is one of *fd*, *lpopt*, and *htd*. For simplicity, we omit the *original* flag and refer to this combination as `gringo`. Table 6.3 presents the number of grounded instances for these four combinations. In the Domain column, we list the domains with their number of tasks. Note that all planning tasks in one domain share the same Datalog rules but the amount of facts changes. In particular, these tasks are increasingly harder. We separate the results into two blocks: including action predicates and excluding them. In each block, we present the number of instances grounded by `gringo`, `gringo+fd`, `gringo+lpopt`, and `gringo+htd`, where we denote them by their initials. For each block, we bold-mark the highest number of grounded instances per domain. In the last row, we sum the results over all domains.

Let us first discuss the "Action Predicates" block. We see that no algorithm managed to ground all 862 tasks. Moreover, none of them is clearly superior to the others. Even though `gringo+lpopt` grounds the most tasks (800), `gringo` and `gringo+htd` ground a similar number. Only `gringo+fd` grounds slightly fewer tasks than the others (775). We notice a considerable difference in the *rovers* domain, where `gringo+fd` manages to ground only 22 tasks within the given timeout while other algorithms ground all 40 tasks. As already noticed in [87], project-join decompositions struggle to produce good decompositions for *rovers*. This case is one example where the heuristic of greedily joining predicates that share the most variables fails. The fact that plain `gringo` is the runner-up might suggest that decomposing Datalog programs is ineffective. Nonetheless, the situation changes when considering programs without action predicates.

We recall that given an action schema $A \in \mathcal{A}$ with $\mathit{vars}(A) = \mathbf{t}$ and precondition $\mathit{pre}(A) = \{p_1(\mathbf{t_1}), \ldots, p_m(\mathbf{t_m})\}$, we generate an *action applicability rule*

$$A\text{-}\mathtt{applicable}(\mathbf{t}) \leftarrow p_1(\mathbf{t_1}), \ldots, p_m(\mathbf{t_m}).$$

and for each atom $q(\mathbf{t}') \in \mathit{add}(A)$ we generate the *effect rule*

$$q(\mathbf{t}') \leftarrow A\text{-}\mathtt{applicable}(\mathbf{t}).$$

where $\mathbf{t} = \bigcup_i \mathbf{t_i}$ and $\mathbf{t}' \subseteq \mathbf{t}$. We refer to A-`applicable`$(\mathbf{t})$ as *action predicate*.

Since an action predicate contains all variables occurring in a rule, we immediately see how their presence influences all the examined methods. Intuitively, a predicate with many variables is more difficult to ground because it has an exponential number of instantiations. When it comes to *tw* and *ghw* they play an additional role. Given a rule r, the action predicate A-`applicable`$(\mathbf{t})$ of r is represented in the primal graph G_r as a clique over the variables in $\mathbf{t}$ and clique vertices must appear together in a bag of any tree decomposition of G_r [25]. Consequently, if G_r has a clique of $|V|$ vertices, $tw(G_r) \geq |V| - 1$. Conversely, the same action predicate A-`applicable`$(\mathbf{t})$ is represented as a single edge over $\mathbf{t}$ in the hypergraph H_r of r. Therefore, H_r is trivially acyclic. This is evident in Table 6.2, where *hw* is always 1 when action predicates are present, whereas *tw* is higher. A notable exception occurs in the *genome-edit-distance* domain, where there is a rule of *ghw* 2 even though action predicates are considered.

Our domains are based on the STRIPS language [48], where defining action schemas with effects utilizing variables not appearing in the precondition is disallowed. On the contrary, the tasks in *genome-edit-distance* are defined in the ADL language [114], where this is possible. However, this has no consequences on the examined methods.

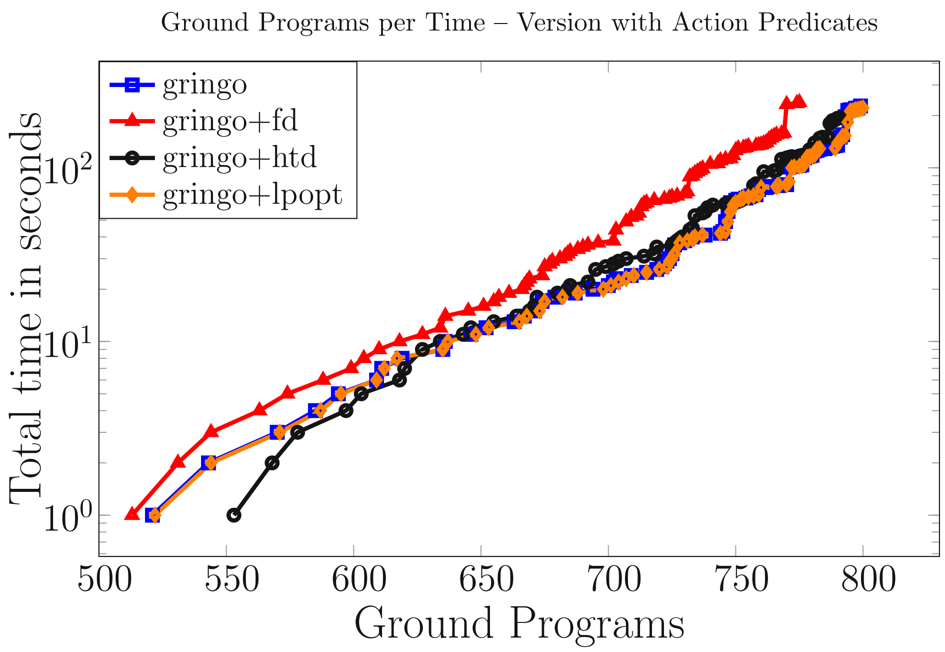
Action predicates are not strictly necessary for the grounding phase. Nevertheless, most state-of-the-art planners use them to speed up the search for a plan. We suspected that action predicates have dramatic consequences on the grounding time. Therefore, we decided to remove them and compare the performances of Datalog rewritings. If necessary, these predicates can be computed later by simply joining the relevant atoms according to the precondition of the rule. Given an action schema $A \in \mathcal{A}$ with $vars(A) = \mathbf{t}$ and precondition $pre(A) = \{p_1(\mathbf{t_1}), \ldots, p_m(\mathbf{t_m})\}$, we avoid generating action applicability rules by explicitly repeating the precondition for each effect atom in $add(A)$

$$\begin{gathered} \mathit{eff}_1(\mathbf{t'_1}) \leftarrow p_1(\mathbf{t_1}), \ldots, p_m(\mathbf{t_m}). \\ \vdots \\ \mathit{eff}_\ell(\mathbf{t'_\ell}) \leftarrow p_1(\mathbf{t_1}), \ldots, p_m(\mathbf{t_m}). \end{gathered}$$

where $\mathbf{t'_j} \subseteq \bigcup_i \mathbf{t_i}$. We see in Table 6.2 that Datalog programs without action predicates have fewer rules than their counterpart. In particular, the simplified program has one rule less for each task action schema. We also notice the opposite effect that removing these predicates has on treewidth and generalized hypertree width. While *tw* decreases, *ghw* increases. Indeed, the treewidth lower bound based on clique size is lower because big cliques disappear from the primal graph. On the contrary, cyclic structures previously hidden in the hypergraph now emerge, leading to higher generalized hypertree width.

Moreover, removing action predicates affects the number of grounded instances, as shown in Table 6.3. All algorithms benefited from the removal as they grounded more tasks than before. Notably, `gringo+fd` shows the highest relative increase $(9,55\%)$ in the number of grounded tasks followed by `gringo+lpopt` $(7,75\%)$, `gringo+htd` $(6,94\%)$, and `gringo` (6%). This is mostly due to the *rovers* tasks, which were all grounded by `gringo+fd` in this experiment. The algorithm that grounded more tasks is once again `gringo+lpopt` with 862 grounded tasks, while plain `gringo` has the worst performance (847). Even though there is a greater difference in contrast to the case where action predicates are present, this consists of only 15 instances. The utility of decompositions remains questionable, but a closer look at the grounding times reveals a different story.

Figure 6.2 relates the number of ground instances with grounding time in seconds for each algorithm. The *y*-axis follows a logarithmic scale. We plot this relation for both the "Action Predicates" and "No Action Predicates" cases. In the first case, we notice that all algorithms perform similarly. Interestingly, not only `gringo` and `gringo+lpopt` ground almost the same number of tasks (799 versus 800), but they also do it with similar running times. There are, however, slight variations that are imperceptible in the plot. It seems that `lpopt` does not decompose most of the rules: only $0,7\%$ of them are decomposed. We attribute this behavior to the presence of action predicates, which



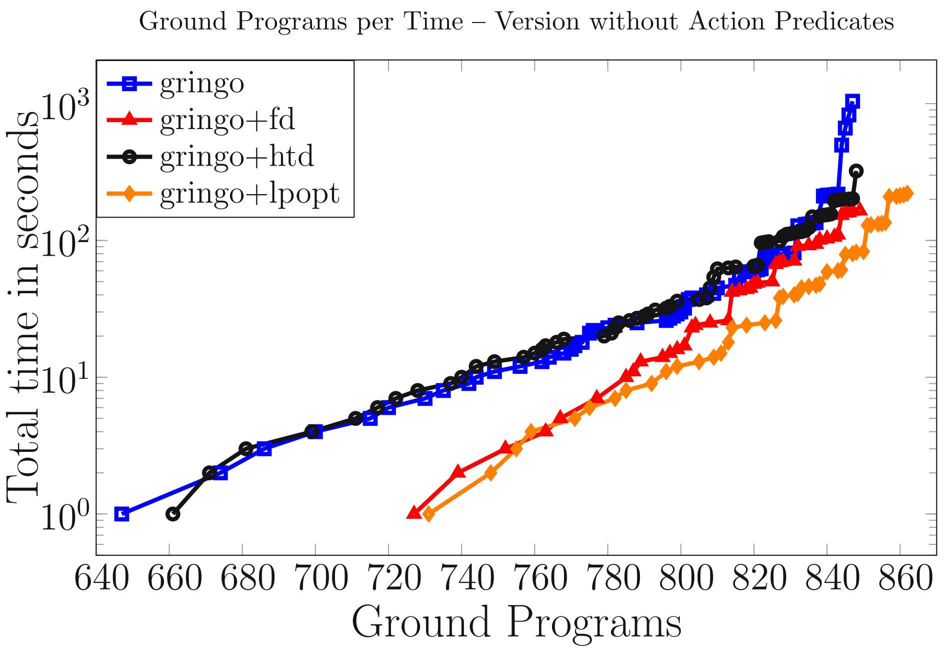


Figure 6.2: Number of ground programs per grounding time (in seconds).

Table 6.4: Average grounding runtimes per domain for different Datalog rewritings. The average is computed as the geometric mean of the grounding time of the tasks that completed within 30 minutes. We abbreviate `gringo` with `G`, `lpopt` with `L`, and `htd` with `H`. Next to each domain we write the number of planning tasks. We separate the results into two blocks according to the presence or absence of action predicates. We mark in bold the best result for each block and domain.

Domain	**Action Predicates**		**No Action Predicates**	
	`G+L`	`G+H`	`G+L`	`G+H`
logistics (40)	**8.93**	20.87	**0.87**	13.50
rovers (40)	10.24	**0.45**	0.41	**0.40**
Total (80)	9.55	**3.18**	**0.60**	2.37

hinder the tree decomposition algorithm with their high number of variables. On the contrary, 100% of the rules are decomposed in the *htd* Datalog programs. Apparently, `BalancedGo` decomposes the rules despite the action predicates. The effect of the rules split is seen on the "Action Predicates" plot in Figure 6.2. While in the majority of the cases it leads to equal or slightly worse performance as `gringo`, the *htd* split is extremely successful on the *rovers* instances, which are the fastest in the plot. The situation drastically changes when considering the "No Action Predicates" case. Figure 6.2 shows that decomposing the Datalog programs results in lower grounding times. This fact is evident when looking at `gringo+lpopt`, which is now clearly superior to plain `gringo`. The `lpopt` program now decomposes about 64% of the rules. While also `gringo+fd` benefits from removing action predicates, `gringo+htd` suffers from it. Intuitively, this makes sense. The *htd* method was the only that could better handle action predicates and was thus favored. However, without action predicates, the hypergraphs of the rules become more cyclic, while the cliques in their primal graphs become smaller. After all, *hw* increases when action predicates are removed whereas *tw* decreases, as seen in Table 6.2.

Altogether, these results leave us with some dismay. Technically, each TD is also a GHD. Theoretically, GHDs have lower widths and, thus, sharper upper bounds on the grounding time. Yet our experiments showed that the TDs computed by `lpopt` lead to much lower grounding times than the GHDs computed by `BalancedGo`. We further investigated this issue by examining the two domains where `gringo+htd` performed best and worst: *rovers* and *logistics*. The running times of other domains are quite homogeneous and not informative. Table 6.4 shows the average grounding times of `gringo+htd` and `gringo+lpopt` on *rovers* and *logistics*. The results are split again into two blocks indicating the presence or absence of action predicates. We compute the running times as a geometric mean of the running times of the domain tasks. When action predicates are present, `gringo+htd` is faster on average thanks to the runtimes achieved on the *rovers* domain. This is because *htd* contains some split rules, while *lpopt* does not. On the other hand, when we remove action predicates and `lpopt` decomposes some rules,

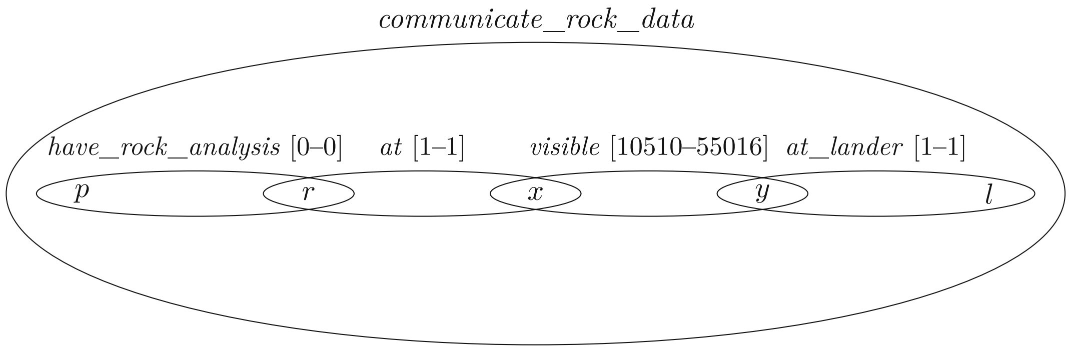


Figure 6.3: Hypergraph of Rule 6.1. For each edge, we indicate the range of facts contained in the initial state across the domain.

`gringo+lpopt` is faster. In this case, the decompositions paint an instructive picture.

In the *rovers* domain, a certain number of rovers navigate a planet's surface to find samples. Their analysis has to be communicated to a lander. The predicate $\mathit{visible}(x, y)$ defines the planet surface by saying that a waypoint y is *visible* from a waypoint x. A rover navigates the planet's surface by moving through sequences of visible waypoints. In a given time instant, the position of a rover r at a waypoint x is encoded by the predicate $\mathit{at}(r, x)$. Likewise, the position of a lander l at a waypoint y is denoted by the predicate $\mathit{at\_lander}(l, y)$. Contrary to a rover, a lander cannot move. Therefore, it will always stay at the same point fixed in the initial state. While navigating the planet, a rover r might collect a rock sample at a waypoint p and thus infer the predicate $\mathit{have\_rock\_analysis}(r, p)$. At this point, a rover r at waypoint x tries to communicate a rock analysis obtained at waypoint p to a lander l located at waypoint y. This action is possible only if y is visible from x. A simplified rule encoding this precondition follows:

$$\begin{aligned}&\mathit{communicate\_rock\_data}(r, l, p, x, y) \leftarrow\\ &\qquad \mathit{at}(r, x), \mathit{at\_lander}(l, y), \mathit{have\_rock\_analysis}(r, p), \mathit{visible}(x, y).\end{aligned} \tag{6.1}$$

Figure 6.3 shows the hypergraph associated with this rule. For each edge name, we report on the range of facts contained in the initial state across the domain instances. Since this hypergraph is small and trivially acyclic, `BalancedGo` computes a width 1 GHD effortlessly. On the contrary, `lpopt` does not decompose the rule leaving it as is. In any case, it is easy to see that the associated primal graph has treewidth 4 due to the action predicate. In this scenario with action predicates, splitting the rule is the best choice, as shown in Table 6.4. In particular, our split is extremely beneficial:

$$\begin{aligned}\mathit{temp}(x, y, l) &\leftarrow \mathit{visible}(x, y), \mathit{at\_lander}(l, y).\\ \mathit{communicate\_rock\_data}(r, l, p, x, y) &\leftarrow \mathit{at}(r, x), \mathit{have\_rock\_analysis}(r, p), \mathit{temp}(x, y, l).\end{aligned}$$

The relation associated with the $\mathit{visible}(x, y)$ predicate is huge because it encodes the whole planet's surface. Likewise, even if the rover starts in one waypoint o, the $\mathit{at}(r, x)$

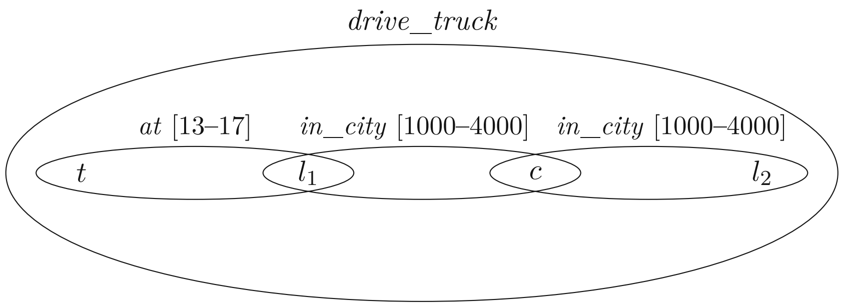


Figure 6.4: Hypergraph of Rule 6.2. For each edge, we indicate the range of facts contained in the initial state across the domain.

relation will eventually contain a tuple for each waypoint reachable from o. This number might get as big as the number of waypoints on the planet. On the other hand, the $\mathit{temp}(x, y, l)$ predicate encodes all positions from which the lander can be seen. Since there is only one lander and it never moves, joining $\mathit{visible}(x, y)$ with $\mathit{at\_lander}(l, y)$ results in filtering a great number of tuples from the $\mathit{visible}(x, y)$ relation. Therefore, in the next step, the join $\mathit{temp}(x, y, l) \bowtie \mathit{at}(r, x)$ can be computed efficiently. Note that even though the relation $\mathit{have\_rock\_analysis}(r, p)$ starts with zero tuples, it ends up having one tuple for each waypoint containing a rock. This number varies across instances between 10–49% of the waypoints, which is always 5499. In case of no action predicates, `lpopt` makes the same splits as `BalancedGo`, and the average grounding times become similar. Even though the runtimes of Table 6.4 are not only due to this specific rule, we observed that similar splits occurred for other rules. Moreover, we implemented a grounder prototype and measured that this is the most triggered rule of the domain and the one where the grounder spends the most time. Indeed, by just splitting this one rule as above and leaving the others intact, `gringo`'s grounding time improves by 95%.

Unfortunately, some splits are detrimental to the grounding phase. In the *logistics* domain, some trucks, airplanes, and parcels are located in one or two cities, each having different locations. The parcels have to be moved by trucks and airplanes to designated locations, with the constraint that while airplanes can fly between cities, trucks can drive only between locations of the same city. Once again, we measured that the following simplified rule is the most challenging of the domain:

$$\mathit{drive\_truck}(t, l_1, l_2, c) \leftarrow \mathit{at}(t, l_1), \mathit{in\_city}(l_1, c), \mathit{in\_city}(l_2, c). \tag{6.2}$$

This rule models the drive of a truck t from a location l_1 to a location t_2 in the same city c. Figure 6.4 shows the hypergraph of this rule. Even in this case, it is trivially acyclic, while the primal graph of the rule has treewidth 3. For the analysis of this case, it is worth knowing that across all instances the number of trucks and cities is at most two whereas the number of locations varies between 1000 and 4000. We observed that

`BalancedGo` splits the rule in the following way:

$$temp(l_1, l_2, c) \leftarrow in\_city(l_1, c), in\_city(l_2, c).$$
$$drive\_truck(t, l_1, l_2, c) \leftarrow at(t, l_1), temp(l_1, l_2, c).$$

This split is poor because the $temp(l_1, l_2, c)$ predicate forces the grounder to compute, for all cities, a cartesian product of the locations. On the contrary, in the case of no action predicates, `lpopt` performs a different split:

$$temp(t, l_1, c) \leftarrow at(t, l_1), in\_city(l_1, c).$$
$$drive\_truck(t, l_1, l_2, c) \leftarrow in\_city(l_2, c), temp(t, l_1, c).$$

In this case, the $temp(t, l_1, c)$ predicate computes the locations l_1 reachable by a truck t in the same city c, which results in filtering out several unreachable locations from $in\_city(l_1, c)$. This split proves better, as seen in Table 6.4.

In conclusion, these two examples demonstrate that even though rules have low *tw* and *ghw*, using the width as the only parameter for quality assessment is unsatisfactory. On the contrary, considering quantitative measures when computing a decomposition substantially improves performance. In particular, join selectivity seemed decisive for the performance of `gringo+htd`. In the *rovers* domain, choosing to perform early a highly selective join allows `gringo+htd` to be the best grounding method among the analyzed ones. On the other hand, forcing the grounder to materialize what essentially is a cartesian product in *logistics* makes `gringo+htd` pay a prohibitive price.

6.4 Summary

In this chapter, we studied the problem of grounding Datalog programs for classical planning. While recapitulating the essential concepts in planning, we emphasized how state-of-the-art planners need to ground a planning task before searching for a solution. In particular, we showed how the grounded task is obtained by formulating a Datalog program corresponding to a relaxed version of the task where no negation occurs.

Since grounding is expensive, we devised a novel method using GHDs to split Datalog rules and produce a new Datalog program equivalent to the original one but easier to ground. We compared our approach against similar decomposition techniques on a well-known benchmark of planning tasks. The results of our experiments showed that, contrary to our expectations, a similar method based on TDs grounds the most tasks in a smaller amount of time on average. This result was counterintuitive because the *ghw* of all Datalog rules is much lower than their *tw*.

Finally, we investigated the reasons for this unexpected behavior. We looked into two significant cases: one where our approach outperformed the competitors and one where it performed the worst. This analysis revealed that executing highly selective joins early and avoiding unnecessary expensive joins is key to obtaining an effective rule decomposition. Therefore, using the width as the only parameter for assessing the decomposition's quality is unsatisfactory. Quantitative measures such as join selectivity are essential.

CHAPTER 7

Conclusion

In this final chapter, we recapitulate our main contributions by putting them in perspective with the state of the field before our work. Additionally, we indicate directions for further research based on our work.

7.1 Contributions

We opened this work in Chapter 3 by tackling the problem of computing GHDs for tractable cases. Combining previously proposed ideas, we formalized the `BalSep` algorithm computing GHDs of width $\leq k$ for the tractable case of hypergraphs having bounded intersection size. Furthermore, we made up for the shortcomings of previous work by proving that `BalSep` is sound and complete.

This new algorithm brings into practice the tractability results obtained by Fischl, Gottlob, and Pichler [53] for the computation of GHDs. Our formal analysis makes `BalSep` a reliable tool for decomposing hypergraphs.

We proceeded in Chapter 4 with an investigation on how to benchmark decomposition algorithms. While turning our attention to *HyperBench*, a preexisting collection of hypergraphs stemming from real-world CQs and CSPs, we recognized that the dataset composition was non-representative of the query-answering realm. Therefore, we rebalanced the partition between CQ and CSP hypergraphs by adding new complex queries with relevant features. Since examining these complex queries was nontrivial, we devised a novel method to extract hypergraphs from non-purely conjunctive queries by identifying their "maximal conjunctive components". Eventually, we studied the structural properties of the extended HyperBench and compared `BalSep` against other decomposition algorithms. We showed that width and (multi-)intersection sizes are small enough to use decomposition algorithms in practice, even for complex queries. Moreover, we observed that `BalSep` is the fastest algorithm for computing GHDs on average.

This study further motivates the use of decompositions in practical problems. Our comparison proved that not only `BalSep` can decompose real-world hypergraphs, but it is also an efficient algorithm. The extension of HyperBench also encourages the development of new algorithms since it is now a representative dataset for comparison. Our novel method for hypergraph extraction from complex queries pushes further the limits of the applicability of decomposition algorithms.

In Chapter 5, we studied the problem of recomputing a GHD upon hypergraph modifications. After defining typical hypergraph modifications, we formalized the SearchUpdateGHD problem. We proved that, for most modifications, SearchUpdateGHD is not polynomial-time solvable. Thus, in theory, knowing a decomposition of the original hypergraph does not make the computation of a GHD for the modified hypergraph easier. Nevertheless, we tackled the problem from a practical perspective proposing a new algorithm that, given as input a decomposition $\mathcal{D}$ of the original hypergraph H and a modified hypergraph H', uses knowledge inferred from $\mathcal{D}$ for quickly computing a new decomposition of H'. Our empirical evaluation showed that our algorithm has speed-up factors between 6 and 50 over the reference algorithm.

The definition of the SearchUpdateGHD opens the door to using decompositions in a previously unexplored setting. That is a dynamic scenario where an instance is repeatedly modified while computing a solution. The results obtained with our prototype suggest that the problem might be easier than the theory establishes.

We finally moved our focus from computing decompositions to using them in Chapter 6, where we use GHDs to ground classical planning tasks. We proposed a novel GHD-guided method to split the rules of a logic program corresponding to a planning task. Through empirical assessment, we showed that rules decompositions help reduce the grounding time, regardless of the type of decomposition used. Nevertheless, we observed that quantitative parameters are equally important, if not more, than structural parameters.

Even though our study shows that using decompositions reduces grounding time, it reveals that existing algorithms do not always produce “good” decompositions. While they aim at computing low-width decompositions, they overlook other quantitative measures. Our experiments suggest that join selectivity is essential to solving a specific instance efficiently. Nevertheless, a more thorough study of other decisive parameters and a quantitative analysis of how they improve performances is necessary.

7.2 Outlook

Our investigation answered many of the open questions posed at the beginning of this thesis. Yet, it raises new ones that we illustrate here.

The first line of research concerns computing decompositions. While our `BalSep` algorithm efficiently computes GHDs in case of bounded intersection size, the tractability results of Fischl, Gottlob, and Pichler [53] also hold for bounded multi-intersection size. Since our work on HyperBench showed that most hypergraphs enjoy small multi-intersection size, new algorithms exploiting this parameter could be developed.

Furthermore, Gottlob et al. [69] reduce the CHECK(fhw, k) problem to CHECK(ghw, k) for classes of hypergraphs of bounded multi-intersection width. However, no efficient implementation of this reduction is available. This line of research could be rewarding.

The study on HyperBench revealed that hypergraphs stemming from real-world CQs and CSPs have favorable properties for decomposition algorithms. Yet, decompositions find their use also in other domains, as our research on grounding planning tasks revealed. Likewise, hypergraphs appear also in *probabilistic graphical models* [14] and *combinatorial auctions* [63]. Collecting and analyzing instances from these domains might provide additional insight into using structural decomposition methods.

In defining the SEARCHUPDATEGHD problem, we wanted to capture the most typical hypergraph modifications. The complexity study revealed that this problem is hard to solve for these classes. Nevertheless, our algorithm for GHD recomputation achieves tremendous speed-ups over classical GHD algorithms. These results suggest that the problem is much easier in practice than in theory. Restricting the type of allowed modifications might lead to discovering some tractable cases.

A second line of research concerns using decompositions to solve database and AI problems. Our work regarding grounding classical planning tasks showed that a decomposition-guided split of logic program rules is enormously advantageous if it considers the quantitative parameters of the instance. On the other hand, splitting rules naively could seriously hinder the grounding performance instead. While we mainly focused on computing low-width decompositions, it became clear that low width is insufficient to assess a decomposition's goodness. On the contrary, quantitative measures such as join selectivity seemed more effective than width alone for grounding. The effects of other parameters such as *join width* [55] remain unknown. Moreover, questions concerning the effective improvement such parameters can achieve remain unanswered. Therefore, we deem urgent an investigation into the parameters making decompositions useful for grounding, answering CQs, and solving CSPs.

Moreover, it is insufficient to understand the parameters leading to a decomposition's success: algorithms computing successful decompositions are necessary. Scarcello et al. [124] proposed an algorithm that, given a hypergraph and a tree aggregating function ϕ, outputs an HD of width $\leq k$ maximizing ϕ. However, this algorithm performs an exhaustive search, which is impractical for larger hypergraphs such as the ones stemming from CSPs. A more scalable approach would be desirable, even if just heuristic.

Systems effectively utilizing hypertree decompositions are missing to date. For instance, in grounding planning tasks, we first decomposed the corresponding logic program and then gave it as input to an external grounder. Splitting the rules requires the definition of unnecessary temporary predicates unused by the successive search for a plan. Unfortunately, the number of temporary predicates is an additional obstacle for grounders because they must instantiate them. Integrating hypertree decompositions directly in the grounder would solve this problem and improve overall performance.

In principle, we could use the same idea to answer queries on databases. Indeed, mainstream database systems struggle to find good join plans for queries joining dozens of tables [105]. On the other hand, GHDs provide a flexible way of splitting such queries, especially if they are cyclic. Preliminary work suggests this direction is promising [67].

Finally, we could also solve CSPs using hypertree decompositions. In this case, the challenge lies in the mix of extensional and intensional constraints. While the firsts are represented in relational form, the latter is defined through logical formulas. Explicitly computing a relation has an exponential cost, in general. Therefore, classical GHDs are not powerful enough to deal with these constraints. Developing smarter algorithms dealing with these constraints without instantiating them is deemed necessary.

List of Figures

List of Tables

List of Algorithms

Bibliography

[1] Blocks world example. `https://ai.dmi.unibas.ch/_files/teaching/hs22/po/slides/po-b01.pdf`. Accessed: 2023-04-23.

[2] C. R. Aberger, A. Lamb, K. Olukotun, and C. Ré. Levelheaded: A unified engine for business intelligence and linear algebra querying. In *34th IEEE International Conference on Data Engineering, ICDE 2018, Paris, France, April 16-19, 2018*, pages 449–460. IEEE Computer Society, 2018.

[3] C. R. Aberger, A. Lamb, S. Tu, A. Nötzli, K. Olukotun, and C. Ré. Emptyheaded: A relational engine for graph processing. *ACM Trans. Database Syst.*, 42(4):20:1–20:44, 2017.

[4] C. R. Aberger, S. Tu, K. Olukotun, and C. 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, Helsinki, Finland, May 16-20, 2016*, pages 97–102. IEEE Computer Society, 2016.

[5] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[6] I. Adler. Marshals, monotone marshals, and hypertree-width. *J. Graph Theory*, 47(4):275–296, 2004.

[7] I. Adler, G. Gottlob, and M. Grohe. Hypertree width and related hypergraph invariants. *Eur. J. Comb.*, 28(8):2167–2181, 2007.

[8] F. N. Afrati, M. R. Joglekar, C. Ré, S. Salihoglu, and J. D. Ullman. GYM: A multiround distributed join algorithm. In M. Benedikt and G. Orsi, editors, *20th International Conference on Database Theory, ICDT 2017, March 21-24, 2017, Venice, Italy*, volume 68 of *LIPIcs*, pages 4:1–4:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017.

[9] S. Agarwal, B. Mozafari, A. Panda, H. Milner, S. Madden, and I. Stoica. Blinkdb: queries with bounded errors and bounded response times on very large data. In Z. Hanzálek, H. Härtig, M. Castro, and M. F. Kaashoek, editors, *Eighth Eurosys Conference 2013, EuroSys '13, Prague, Czech Republic, April 14-17, 2013*, pages 29–42. ACM, 2013.

[10] R. Agrawal, M. J. Carey, and M. Livny. Concurrency control performance modeling: Alternatives and implications. *ACM Trans. Database Syst.*, 12(4):609–654, 1987.

[11] K. Amroun, Z. Habbas, and W. Aggoune-Mtalaa. A compressed generalized hypertree decomposition-based solving technique for non-binary constraint satisfaction problems. *AI Commun.*, 29(2):371–392, 2016.

[12] M. Aref, B. ten Cate, T. J. Green, B. Kimelfeld, D. Olteanu, E. Pasalic, T. L. Veldhuizen, and G. Washburn. Design and implementation of the logicblox system. In T. K. Sellis, S. B. Davidson, and Z. G. Ives, editors, *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, pages 1371–1382. ACM, 2015.

[13] P. C. Arocena, B. Glavic, R. Ciucanu, and R. J. Miller. The ibench integration metadata generator. *PVLDB*, 9(3):108–119, 2015.

[14] A. S. Arun, S. V. M. Jayaraman, C. Ré, and A. Rudra. Hypertree decompositions revisited for pgms. *CoRR*, abs/1807.00886, 2018.

[15] G. Audemard, F. Boussemart, C. Lecoutre, and C. Piette. XCSP3: an XML-based format designed to represent combinatorial constrained problems, 2016.

[16] N. Bakibayev, T. Kociský, D. Olteanu, and J. Zavodny. Aggregation and ordering in factorised databases. *PVLDB*, 6(14):1990–2001, 2013.

[17] C. Baral. *Knowledge Representation, Reasoning and Declarative Problem Solving*. Cambridge University Press, 2010.

[18] P. Beame, S. A. Cook, J. Edmonds, R. Impagliazzo, and T. Pitassi. The relative complexity of NP search problems. *J. Comput. Syst. Sci.*, 57(1):3–19, 1998.

[19] C. Beeri, R. Fagin, D. Maier, and M. Yannakakis. On the desirability of acyclic database schemes. *J. ACM*, 30(3):479–513, 1983.

[20] M. Benedikt. CQ benchmarks, 2017. Personal Communication.

[21] M. Benedikt, G. Konstantinidis, G. Mecca, B. Motik, P. Papotti, D. Santoro, and E. Tsamoura. Benchmarking the chase. In E. Sallinger, J. V. den Bussche, and F. Geerts, editors, *Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS 2017, Chicago, IL, USA, May 14-19, 2017*, pages 37–52. ACM, 2017.

[22] J. Berg, N. Lodha, M. Järvisalo, and S. Szeider. Maxsat benchmarks based on determining generalized hypertree-width. *MaxSAT Evaluation 2017: Solver and Benchmark Descriptions*, B-2017-2:22, 2017.

[23] M. Bichler, M. Morak, and S. Woltran. lpopt: A rule optimization tool for answer set programming. *Fundam. Informaticae*, 177(3-4):275–296, 2020.

[24] H. L. Bodlaender. A linear-time algorithm for finding tree-decompositions of small treewidth. *SIAM J. Comput.*, 25(6):1305–1317, 1996.

[25] H. L. Bodlaender and A. M. Koster. Treewidth computations ii. lower bounds. *Information and Computation*, 209(7):1103–1119, 2011.

[26] B. Bonet and H. Geffner. Planning as heuristic search. *Artif. Intell.*, 129(1-2):5–33, 2001.

[27] A. Bonifati, W. Martens, and T. Timm. An analytical study of large SPARQL query logs. *PVLDB*, 11(2):149–161, 2017.

[28] A. Bonifati, W. Martens, and T. Timm. Navigating the maze of wikidata query logs. In L. Liu, R. W. White, A. Mantrach, F. Silvestri, J. J. McAuley, R. Baeza-Yates, and L. Zia, editors, *The World Wide Web Conference, WWW 2019, San Francisco, CA, USA, May 13-17, 2019*, pages 127–138. ACM, 2019.

[29] K. E. C. Booth, T. T. Tran, G. Nejat, and J. C. Beck. Mixed-integer and constraint programming techniques for mobile robot task planning. *IEEE Robotics Autom. Lett.*, 1(1):500–507, 2016.

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

[31] J. Brault-Baron. Hypergraph acyclicity revisited. *ACM Comput. Surv.*, 49(3):54:1–54:26, 2016.

[32] T. Bylander. Complexity results for planning. In J. Mylopoulos and R. Reiter, editors, *Proceedings of the 12th International Joint Conference on Artificial Intelligence. Sydney, Australia, August 24-30, 1991*, pages 274–279. Morgan Kaufmann, 1991.

[33] F. Calimeri, S. Perri, and J. Zangari. Optimizing answer set computation via heuristic-based decomposition. *Theory Pract. Log. Program.*, 19(4):603–628, 2019.

[34] S. Ceri and J. Widom. Deriving production rules for incremental view maintenance. In G. M. Lohman, A. Sernadas, and R. Camps, editors, *17th International Conference on Very Large Data Bases, September 3-6, 1991, Barcelona, Catalonia, Spain, Proceedings*, pages 577–589. Morgan Kaufmann, 1991.

[35] A. K. Chandra and P. M. Merlin. Optimal implementation of conjunctive queries in relational data bases. In J. E. Hopcroft, E. P. Friedman, and M. A. Harrison, editors, *Proceedings of the 9th Annual ACM Symposium on Theory of Computing, May 4-6, 1977, Boulder, Colorado, USA*, pages 77–90. ACM, 1977.

[36] C. Chekuri and A. Rajaraman. Conjunctive query containment revisited. In F. N. Afrati and P. G. Kolaitis, editors, *Database Theory - ICDT '97, 6th International Conference, Delphi, Greece, January 8-10, 1997, Proceedings*, volume 1186 of *Lecture Notes in Computer Science*, pages 56–70. Springer, 1997.

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

[38] A. B. Corrêa, M. Hecher, M. Helmert, D. M. Longo, F. Pommerening, and S. Woltran. Grounding planning tasks using tree decompositions and iterated solving. In *Proceedings of the Thirty-Third International Conference on Automated Planning and Scheduling, ICAPS 2023, Prague, July 8-13, 2023*, 2023.

[39] A. B. Corrêa, F. Pommerening, M. Helmert, and G. Francès. Lifted successor generation using query optimization techniques. In J. C. Beck, O. Buffet, J. Hoffmann, E. Karpas, and S. Sohrabi, editors, *Proceedings of the Thirtieth International Conference on Automated Planning and Scheduling, Nancy, France, October 26-30, 2020*, pages 80–89. AAAI Press, 2020.

[40] E. Dantsin, T. Eiter, G. Gottlob, and A. Voronkov. Complexity and expressive power of logic programming. *ACM Comput. Surv.*, 33(3):374–425, 2001.

[41] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, 2008.

[42] R. Dechter. *Constraint processing.* Elsevier Morgan Kaufmann, 2003.

[43] R. Dechter and J. Pearl. Tree clustering for constraint networks. *Artif. Intell.*, 38(3):353–366, 1989.

[44] K. Erol, D. S. Nau, and V. S. Subrahmanian. Complexity, decidability and undecidability results for domain-independent planning. *Artif. Intell.*, 76(1-2):75–88, 1995.

[45] R. Fagin. Degrees of acyclicity for hypergraphs and relational database schemes. *J. ACM*, 30(3):514–550, 1983.

[46] B. Falkenhainer and K. D. Forbus. Compositional modeling: Finding the right model for the job. *Artif. Intell.*, 51(1-3):95–143, 1991.

[47] J. K. Fichte, M. Hecher, N. Lodha, and S. Szeider. An SMT approach to fractional hypertree width. In J. N. Hooker, editor, *Principles and Practice of Constraint Programming - 24th International Conference, CP 2018, Lille, France, August 27-31, 2018, Proceedings*, volume 11008 of *Lecture Notes in Computer Science*, pages 109–127. Springer, 2018.

[48] R. Fikes and N. J. Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. *Artif. Intell.*, 2(3/4):189–208, 1971.

[49] W. Fischl. *Generalized and Fractional Hypertree Decompositions From Theory to Practice.* PhD thesis, TU Wien, 2018.

[50] W. Fischl, G. Gottlob, D. M. Longo, and R. Pichler. Hyperbench: A benchmark and tool for hypergraphs and empirical findings. In D. Suciu, S. Skritek, and C. Koch, editors, *Proceedings of the 38th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*, pages 464–480. ACM, 2019.

[51] W. Fischl, G. Gottlob, D. M. Longo, and R. Pichler. Hyperbench: A benchmark and tool for hypergraphs and empirical findings. In A. Hogan and T. Milo, editors, *Proceedings of the 13th Alberto Mendelzon International Workshop on Foundations of Data Management, Asunción, Paraguay, June 3-7, 2019*, volume 2369 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2019.

[52] W. Fischl, G. Gottlob, D. M. Longo, and R. Pichler. Hyperbench: A benchmark and tool for hypergraphs and empirical findings. *ACM J. Exp. Algorithmics*, 26:1.6:1–1.6:40, 2021.

[53] W. Fischl, G. Gottlob, and R. Pichler. General and fractional hypertree decompositions: Hard and easy cases. In J. V. den Bussche and M. Arenas, editors, *Proceedings of the 37th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, Houston, TX, USA, June 10-15, 2018*, pages 17–32. ACM, 2018.

[54] B. N. Freeman-Benson, J. Maloney, and A. Borning. An incremental constraint solver. *Commun. ACM*, 33(1):54–63, 1990.

[55] R. Ganian, S. Ordyniak, and S. Szeider. A join-based hybrid parameter for constraint satisfaction. In T. Schiex and S. de Givry, editors, *Principles and Practice of Constraint Programming - 25th International Conference, CP 2019, Stamford, CT, USA, September 30 - October 4, 2019, Proceedings*, volume 11802 of *Lecture Notes in Computer Science*, pages 195–212. Springer, 2019.

[56] R. Ganian, A. Schidler, M. Sorge, and S. Szeider. Threshold treewidth and hypertree width. *J. Artif. Intell. Res.*, 74:1687–1713, 2022.

[57] T. Ganzow, G. Gottlob, N. Musliu, and M. Samer. A csp hypergraph library. techreport, TU Wien, 2005.

[58] M. Gebser, R. Kaminski, B. Kaufmann, and T. Schaub. Multi-shot ASP solving with clingo. *Theory Pract. Log. Program.*, 19(1):27–82, 2019.

[59] M. Gebser, B. Kaufmann, A. Neumann, and T. Schaub. *clasp* : A conflict-driven answer set solver. In C. Baral, G. Brewka, and J. S. Schlipf, editors, *Logic Programming and Nonmonotonic Reasoning, 9th International Conference, LPNMR 2007, Tempe, AZ, USA, May 15-17, 2007, Proceedings*, volume 4483 of *Lecture Notes in Computer Science*, pages 260–265. Springer, 2007.

[60] F. Geerts, G. Mecca, P. Papotti, and D. Santoro. Mapping and cleaning. In I. F. Cruz, E. Ferrari, Y. Tao, E. Bertino, and G. Trajcevski, editors, *IEEE 30th International Conference on Data Engineering, Chicago, ICDE 2014, IL, USA, March 31 - April 4, 2014*, pages 232–243. IEEE Computer Society, 2014.

[61] L. Ghionna, L. Granata, G. Greco, and F. Scarcello. Hypertree decompositions for query optimization. In R. Chirkova, A. Dogac, M. T. Özsu, and T. K. Sellis, editors, *Proceedings of the 23rd International Conference on Data Engineering, ICDE 2007, The Marmara Hotel, Istanbul, Turkey, April 15-20, 2007*, pages 36–45. IEEE Computer Society, 2007.

[62] L. Ghionna, G. Greco, and F. Scarcello. H-DB: a hybrid quantitative-structural sql optimizer. In C. Macdonald, I. Ounis, and I. Ruthven, editors, *Proceedings of the 20th ACM Conference on Information and Knowledge Management, CIKM 2011, Glasgow, United Kingdom, October 24-28, 2011*, pages 2573–2576. ACM, 2011.

[63] G. Gottlob and G. Greco. Decomposing combinatorial auctions and set packing problems. *J. ACM*, 60(4):24:1–24:39, 2013.

[64] G. Gottlob, G. Greco, N. Leone, and F. Scarcello. Hypertree decompositions: Questions and answers. In T. Milo and W. Tan, editors, *Proceedings of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, pages 57–74. ACM, 2016.

[65] G. Gottlob, M. Lanzinger, D. M. Longo, and C. Okulmus. Incremental updates of generalized hypertree decompositions. *ACM J. Exp. Algorithmics*, 27, mar 2023.

[66] G. Gottlob, M. Lanzinger, D. M. Longo, C. Okulmus, and R. Pichler. The hypertrac project: Recent progress and future research directions on hypergraph decompositions. In E. Hebrard and N. Musliu, editors, *Integration of Constraint Programming, Artificial Intelligence, and Operations Research - 17th International Conference, CPAIOR 2020, Vienna, Austria, September 21-24, 2020, Proceedings*, volume 12296 of *Lecture Notes in Computer Science*, pages 3–21. Springer, 2020.

[67] G. Gottlob, M. Lanzinger, D. M. Longo, C. Okulmus, R. Pichler, and A. Selzer. Structure-guided query evaluation: Towards bridging the gap from theory to practice. *CoRR*, abs/2303.02723, 2023.

[68] G. Gottlob, M. Lanzinger, R. Pichler, and I. Razgon. Complexity analysis of generalized and fractional hypertree decompositions. *CoRR*, abs/2002.05239, 2020.

[69] G. Gottlob, M. Lanzinger, R. Pichler, and I. Razgon. Complexity analysis of generalized and fractional hypertree decompositions. *J. ACM*, 68(5):38:1–38:50, 2021.

[70] G. Gottlob, N. Leone, and F. Scarcello. The complexity of acyclic conjunctive queries. *J. ACM*, 48(3):431–498, 2001.

[71] G. Gottlob, N. Leone, and F. Scarcello. Hypertree decompositions: A survey. In J. Sgall, A. Pultr, and P. Kolman, editors, *Mathematical Foundations of Computer Science 2001, 26th International Symposium, MFCS 2001 Marianske Lazne, Czech Republic, August 27-31, 2001, Proceedings*, volume 2136 of *Lecture Notes in Computer Science*, pages 37–57. Springer, 2001.

[72] G. Gottlob, N. Leone, and F. Scarcello. Hypertree decompositions and tractable queries. *J. Comput. Syst. Sci.*, 64(3):579–627, 2002.

[73] G. Gottlob, Z. Miklós, and T. Schwentick. Generalized hypertree decompositions: Np-hardness and tractable variants. *J. ACM*, 56(6):30:1–30:32, 2009.

[74] G. Gottlob, C. Okulmus, and R. Pichler. Fast and parallel decomposition of constraint satisfaction problems. In C. Bessiere, editor, *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, IJCAI 2020 [scheduled for July 2020, Yokohama, Japan, postponed due to the Corona pandemic]*, pages 1155–1162. ijcai.org, 2020.

[75] G. Gottlob, C. Okulmus, and R. Pichler. Fast and parallel decomposition of constraint satisfaction problems. *Constraints An Int. J.*, 27(3):284–326, 2022.

[76] G. Gottlob and M. Samer. A backtracking-based algorithm for hypertree decomposition. *ACM Journal of Experimental Algorithmics*, 13:1:1.1–1:1.19, 2008.

[77] M. H. Graham. On the universal relation. Technical report, University of Toronto, 1979.

[78] J. Gray, R. A. Lorie, G. R. Putzolu, and I. L. Traiger. Granularity of locks and degrees of consistency in a shared data base. In G. M. Nijssen, editor, *Modelling in Data Base Management Systems, Proceeding of the IFIP Working Conference on Modelling in Data Base Management Systems, Freudenstadt, Germany, January 5-8, 1976*, pages 365–394. North-Holland, 1976.

[79] M. Grohe and D. Marx. Constraint solving via fractional edge covers. *ACM Trans. Algorithms*, 11(1):4:1–4:20, 2014.

[80] Y. Guo, Z. Pan, and J. Heflin. LUBM: A benchmark for OWL knowledge base systems. *J. Web Semant.*, 3(2-3):158–182, 2005.

[81] A. Gupta, I. S. Mumick, and V. S. Subrahmanian. Maintaining views incrementally. In P. Buneman and S. Jajodia, editors, *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data, Washington, DC, USA, May 26-28, 1993*, pages 157–166. ACM Press, 1993.

[82] Z. Habbas, K. Amroun, and D. 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.

[83] V. Harinarayan, A. Rajaraman, and J. D. Ullman. Implementing data cubes efficiently. In H. V. Jagadish and I. S. Mumick, editors, *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data, Montreal, Quebec, Canada, June 4-6, 1996*, pages 205–216. ACM Press, 1996.

[84] P. Haslum et al. Computing genome edit distances using domain-independent planning. 2011.

[85] P. Haslum, N. Lipovetzky, D. Magazzeni, and C. Muise. *An Introduction to the Planning Domain Definition Language*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers, 2019.

[86] M. Helmert. The fast downward planning system. *J. Artif. Intell. Res.*, 26:191–246, 2006.

[87] M. Helmert. Concise finite-domain representations for PDDL planning tasks. *Artif. Intell.*, 173(5-6):503–535, 2009.

[88] J. Hoffmann and B. Nebel. The FF planning system: Fast plan generation through heuristic search. *J. Artif. Intell. Res.*, 14:253–302, 2001.

[89] N. Immerman. Relational queries computable in polynomial time. *Inf. Control.*, 68(1-3):86–104, 1986.

[90] S. Jain, D. Moritz, D. Halperin, B. Howe, and E. Lazowska. Sqlshare: Results from a multi-year sql-as-a-service experiment. In F. Özcan, G. Koutrika, and S. Madden, editors, *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, pages 281–293. ACM, 2016.

[91] S. Karakashian, R. J. Woodward, and B. Y. Choueiry. Reformulating r(*, m)c with tree decomposition. In M. R. Genesereth and P. Z. Revesz, editors, *Proceedings of the Ninth Symposium on Abstraction, Reformulation, and Approximation, SARA 2011, Parador de Cardona, Cardona, Catalonia, Spain, July 17-18, 2011.*, pages 62–69. AAAI, 2011.

[92] M. Katz and J. Hoffmann. Mercury planner: Pushing the limits of partial delete relaxation. *IPC 2014 planner abstracts*, pages 43–47, 2014.

[93] M. A. Khamis, R. R. Curtin, B. Moseley, H. Q. Ngo, X. Nguyen, D. Olteanu, and M. Schleich. Functional aggregate queries with additive inequalities. *ACM Trans. Database Syst.*, 45(4):17:1–17:41, 2020.

[94] M. A. Khamis, H. Q. Ngo, C. Ré, and A. Rudra. Joins via geometric resolutions: Worst case and beyond. *ACM Trans. Database Syst.*, 41(4):22:1–22:45, 2016.

[95] M. A. Khamis, H. Q. Ngo, and A. Rudra. FAQ: questions asked frequently. In T. Milo and W. Tan, editors, *Proceedings of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, pages 13–28. ACM, 2016.

[96] M. A. Khamis, H. Q. Ngo, and D. Suciu. What do shannon-type inequalities, submodular width, and disjunctive datalog have to do with one another? In E. Sallinger, J. V. den Bussche, and F. Geerts, editors, *Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS 2017, Chicago, IL, USA, May 14-19, 2017*, pages 429–444. ACM, 2017.

[97] P. G. Kolaitis and M. Y. Vardi. Conjunctive-query containment and constraint satisfaction. In A. O. Mendelzon and J. Paredaens, editors, *Proceedings of the Seventeenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 1-3, 1998, Seattle, Washington, USA*, pages 205–213. ACM Press, 1998.

[98] P. Laborie, J. Rogerie, P. Shaw, and P. Vilím. IBM ILOG CP optimizer for scheduling - 20+ years of scheduling with constraints at IBM/ILOG. *Constraints An Int. J.*, 23(2):210–250, 2018.

[99] M. Lalou, Z. Habbas, and K. Amroun. Solving hypertree structured CSP: sequential and parallel approaches. In M. Gavanelli and T. Mancini, editors, *Proceedings of the 16th RCRA workshop on Experimental Evaluation of Algorithms for Solving Problems with Combinatorial Explosion, RCRA@AI*IA 2009, Reggio Emilia, Italy, December 11-12, 2009*, volume 589 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2009.

[100] P. Lauer, Á. Torralba, D. Fiser, D. Höller, J. Wichlacz, and J. Hoffmann. Polynomial-time in PDDL input size: Making the delete relaxation feasible for lifted planning. In Z. Zhou, editor, *Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence, IJCAI 2021, Virtual Event / Montreal, Canada, 19-27 August 2021*, pages 4119–4126. ijcai.org, 2021.

[101] V. Leis, B. Radke, A. Gubichev, A. Mirchev, P. A. Boncz, A. Kemper, and T. Neumann. Query optimization through the looking glass, and what we found running the join order benchmark. *VLDB J.*, 27(5):643–668, 2018.

[102] N. Leone, G. Pfeifer, W. Faber, T. Eiter, G. Gottlob, S. Perri, and F. Scarcello. The DLV system for knowledge representation and reasoning. *ACM Trans. Comput. Log.*, 7(3):499–562, 2006.

[103] A. K. Mackworth and E. C. Freuder. The complexity of constraint satisfaction revisited. *Artif. Intell.*, 59(1-2):57–62, 1993.

[104] S. Malyshev, M. Krötzsch, L. González, J. Gonsior, and A. Bielefeldt. Getting the most out of wikidata: Semantic technology usage in wikipedia's knowledge graph. In D. Vrandecic, K. Bontcheva, M. C. Suárez-Figueroa, V. Presutti, I. Celino, M. Sabou, L. Kaffee, and E. Simperl, editors, *The Semantic Web - ISWC 2018 - 17th International Semantic Web Conference, Monterey, CA, USA, October 8-12, 2018, Proceedings, Part II*, volume 11137 of *Lecture Notes in Computer Science*, pages 376–394. Springer, 2018.

[105] R. Mancini, S. Karthik, B. Chandra, V. Mageirakos, and A. Ailamaki. Efficient massively parallel join optimization for large queries. In Z. G. Ives, A. Bonifati, and A. E. Abbadi, editors, *SIGMOD '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022*, pages 122–135. ACM, 2022.

[106] D. Marx. Tractable hypergraph properties for constraint satisfaction and conjunctive queries. *J. ACM*, 60(6):42:1–42:51, 2013.

[107] R. Matloob and M. Soutchanski. Exploring organic synthesis with state-of-the-art planning techniques. In *Proc. SPARK Workshop*, pages 52–61, 2016.

[108] D. Michail, J. Kinable, B. Naveh, and J. V. Sichi. Jgrapht—a java library for graph data structures and algorithms. *ACM Trans. Math. Softw.*, 46(2), May 2020.

[109] C. Mohan, B. G. Lindsay, and R. Obermarck. Transaction management in the r* distributed database management system. *ACM Trans. Database Syst.*, 11(4):378–396, 1986.

[110] M. Morak and S. Woltran. Preprocessing of complex non-ground rules in answer set programming. In A. Dovier and V. S. Costa, editors, *Technical Communications of the 28th International Conference on Logic Programming, ICLP 2012, September 4-8, 2012, Budapest, Hungary*, volume 17 of *LIPIcs*, pages 247–258. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2012.

[111] I. Niemelä and P. Simons. Smodels-an implementation of the stable model and well-founded semantics for normal logic programs. In *Proceedings of the 4th international conference on logic programming and non-monotonic reasoning*, volume 1265, pages 420–429. Springer Berlin, 1997.

[112] D. Olteanu and J. Závodný. Size bounds for factorised representations of query results. *ACM Trans. Database Syst.*, 40(1):2:1–2:44, 2015.

[113] J. Pearson and P. G. Jeavons. A survey of tractable constraint satisfaction problems. Technical report, Technical Report CSD-TR-97-15, Royal Holloway, University of London, 1997.

[114] E. P. D. Pednault. ADL: exploring the middle ground between STRIPS and the situation calculus. In R. J. Brachman, H. J. Levesque, and R. Reiter, editors,

Proceedings of the 1st International Conference on Principles of Knowledge Representation and Reasoning (KR'89). Toronto, Canada, May 15-18 1989, pages 324–332. Morgan Kaufmann, 1989.

[115] A. Perelman and C. Ré. Duncecap: Compiling worst-case optimal query plans. In T. K. Sellis, S. B. Davidson, and Z. G. Ives, editors, *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, pages 2075–2076. ACM, 2015.

[116] J. Pérez, M. Arenas, and C. Gutiérrez. Semantics and complexity of SPARQL. *ACM Trans. Database Syst.*, 34(3):16:1–16:45, 2009.

[117] F. Picalausa and S. Vansummeren. What are real SPARQL queries like? In R. D. Virgilio, F. Giunchiglia, and L. Tanca, editors, *Proceedings of the International Workshop on Semantic Web Information Management, SWIM 2011, Athens, Greece, June 12, 2011*, page 7. ACM, 2011.

[118] R. Pottinger and A. Y. Halevy. Minicon: A scalable algorithm for answering queries using views. *VLDB J.*, 10(2-3):182–198, 2001.

[119] X. Qian. Query folding. In S. Y. W. Su, editor, *Proceedings of the Twelfth International Conference on Data Engineering, February 26 - March 1, 1996, New Orleans, Louisiana, USA*, pages 48–55. IEEE Computer Society, 1996.

[120] L. D. Raedt, T. Guns, and S. Nijssen. Constraint programming for data mining and machine learning. In M. Fox and D. Poole, editors, *Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2010, Atlanta, Georgia, USA, July 11-15, 2010.* AAAI Press, 2010.

[121] S. Richter and M. Westphal. The LAMA planner: Guiding cost-based anytime planning with landmarks. *J. Artif. Intell. Res.*, 39:127–177, 2010.

[122] N. Robertson and P. D. Seymour. Graph minors. III. planar tree-width. *J. Comb. Theory, Ser. B*, 36(1):49–64, 1984.

[123] N. Robertson and P. D. Seymour. Graph minors. II. algorithmic aspects of tree-width. *J. Algorithms*, 7(3):309–322, 1986.

[124] F. Scarcello, G. Greco, and N. Leone. Weighted hypertree decompositions and optimal query plans. *J. Comput. Syst. Sci.*, 73(3):475–506, 2007.

[125] A. Schidler and S. Szeider. Computing optimal hypertree decompositions. In G. E. Blelloch and I. Finocchi, editors, *Proceedings of the Symposium on Algorithm Engineering and Experiments, ALENEX 2020, Salt Lake City, UT, USA, January 5-6, 2020*, pages 1–11. SIAM, 2020.

[126] A. Schidler and S. Szeider. Computing optimal hypertree decompositions with SAT. In Z. Zhou, editor, *Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence, IJCAI 2021, Virtual Event / Montreal, Canada, 19-27 August 2021*, pages 1418–1424. ijcai.org, 2021.

[127] R. Seidel. A new method for solving constraint satisfaction problems. In P. J. Hayes, editor, *Proceedings of the 7th International Joint Conference on Artificial Intelligence, IJCAI '81, Vancouver, BC, Canada, August 24-28, 1981*, pages 338–342. William Kaufmann, 1981.

[128] P. Shaw. Using constraint programming and local search methods to solve vehicle routing problems. In M. J. Maher and J. Puget, editors, *Principles and Practice of Constraint Programming - CP98, 4th International Conference, Pisa, Italy, October 26-30, 1998, Proceedings*, volume 1520 of *Lecture Notes in Computer Science*, pages 417–431. Springer, 1998.

[129] R. E. Tarjan. Depth-first search and linear graph algorithms. *SIAM J. Comput.*, 1(2):146–160, 1972.

[130] R. E. Tarjan and M. Yannakakis. Simple linear-time algorithms to test chordality of graphs, test acyclicity of hypergraphs, and selectively reduce acyclic hypergraphs. *SIAM J. Comput.*, 13(3):566–579, 1984.

[131] Transaction Processing Performance Council (TPC). TPC-DS decision support benchmark, 2006.

[132] Transaction Processing Performance Council (TPC). TPC-H decision support benchmark, 2014.

[133] S. Tu and C. Ré. Duncecap: Query plans using generalized hypertree decompositions. In T. K. Sellis, S. B. Davidson, and Z. G. Ives, editors, *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, pages 2077–2078. ACM, 2015.

[134] J. D. Ullman. *Principles of Database and Knowledge-Base Systems, Volume I*, volume 14 of *Principles of computer science series*. Computer Science Press, 1988.

[135] J. D. Ullman. *Principles of Database and Knowledge-Base Systems, Volume II*. Computer Science Press, 1989.

[136] V. N. Vapnik and A. Y. Chervonenkis. On the uniform convergence of relative frequencies of events to their probabilities. *Theory of Probability & Its Applications*, 16(2):264–280, jan 1971.

[137] M. Y. Vardi. The complexity of relational query languages (extended abstract). In H. R. Lewis, B. B. Simons, W. A. Burkhard, and L. H. Landweber, editors, *Proceedings of the 14th Annual ACM Symposium on Theory of Computing, May 5-7, 1982, San Francisco, California, USA*, pages 137–146. ACM, 1982.

[138] H. Verhaeghe, S. Nijssen, G. Pesant, C. Quimper, and P. Schaus. Learning optimal decision trees using constraint programming. *Constraints An Int. J.*, 25(3-4):226–250, 2020.

[139] T. Warneke. Jsqlparser, 2019.

[140] M. Yannakakis. Algorithms for acyclic database schemes. In *Very Large Data Bases, 7th International Conference, September 9-11, 1981, Cannes, France, Proceedings*, pages 82–94. IEEE Computer Society, 1981.

[141] C. T. Yu and M. Z. Ozsoyoglu. An algorithm for tree-query membership of a distributed query. In *The IEEE Computer Society's Third International Computer Software and Applications Conference, COMPSAC 1979, 6-8 November, 1979, Chicago, Illinois, USA*, pages 306–312. IEEE, 1979.

[142] Y. Yu, M. Isard, D. Fetterly, M. Budiu, Ú. Erlingsson, P. K. Gunda, and J. Currey. Dryadlinq: A system for general-purpose distributed data-parallel computing using a high-level language. In R. Draves and R. van Renesse, editors, *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings*, pages 1–14. USENIX Association, 2008.