Hypertree Decomposition

Victor-Teodor Stoian

4th-Year Project Report
Artificial Intelligence and Computer Science
School of Informatics
University of Edinburgh

2021

Abstract

”Everything nowadays is controlled by algorithms and they either do it too well for you to notice or too poorly to be noticed” can be the description of the quotidian life. We have grown so accustomed to software that it is an essential component of our daily lives and almost all pieces of software use a database in order to persist data. A database is a structure for persisting organised data on an electronic device and it represents one of the main ways of storing large volumes of data for fast retrieval. Thus, their optimization is a necessity in order to keep up with the ever-growing amount of data that we generate (worldwide data is expected to hit 175 zettabytes by 2025 [14])
Among the first structures proposed for a database is the relational database proposed by E. F. Codd in 1970 and is defined by grouping records into 'tables' according to their semantic values and establishing the relationships between the records through 'foreign keys'. When information from multiple tables needs to be retrieved, the software managing the database performs 'joins' on the necessary tables to match the correct records and retrieve the needed information. But, as the reader will see in this report, the language agnostic constraints of the query are equivalent with the Constraint Satisfaction Problem [16] in the form of Conjunctive Queries which is an NP-Complete problem, so no efficient algorithm is known to solve a general CSP.
However, it has been proven that for certain Conjunctive Queries, more specifically for acyclic Conjunctive Queries, there exists an algorithm developed by M.Yannakakis [21] that can compute the result of the query against a database instance in asymptotic polynomial time. Following this result, there have been various attempts of generalizing this property from acyclic queries to 'weak cyclic queries' through query decompositions.

In this work, I follow the pseudo-codes of two novel query decomposition algorithms to critically assess and amend the canonical algorithms, pioneer two of the first implementations in the fields of Hypertree Decomposition and Fractional Hypertree Decomposition and test their practical utility on randomly sampled examples from the benchmarking platform in this domain, Hyperbench [19], as well as examples from existing literature.

Acknowledgements

I want to thank my supervisor, Nikolic Milos for the continuous help and guidance throughout the development of this project.

Acronyms

- **CQ** Conjunctive Query.
- **UCQ** Union of Conjunctive Queries.
- **ACQ** Acyclic Conjunctive Queries.
- **CSP** Constraint Satisfaction Problem.
- **vars(q)** The set of all variables that appear in the body of the Conjunctive Query q.
- **head(q)** The set of all variables that appear in the head of the Conjunctive Query q.
- **td(q)** The tree decomposition of the hypergraph equivalent with the Conjunctive Query q.
- **tw(q)** The tree width of the hypergraph equivalent with the Conjunctive Query q.
- **hd(q)** The hypertree decomposition of the hypergraph equivalent with the Conjunctive Query q.
- **hw(q)** The hypertree width of the hypergraph equivalent with the Conjunctive Query q.
- **ghd(q)** The generalized hypertree decomposition of the hypergraph equivalent with the Conjunctive Query q.
- **ghw(q)** The generalized hypertree width of the hypergraph equivalent with the Conjunctive Query q.
- **fhd(q)** The fractional hypertree decomposition of the hypergraph equivalent with the Conjunctive Query q.
- **fhw(q)** The fractional hypertree width of the hypergraph equivalent with the Conjunctive Query q.

Table of Contents

Chapter 1

Introduction

As data-driven projects become increasingly more popular and the amount of data that we generate is growing at an exponential rate, research into efficient information storage and retrieval from a database is more important than ever. Formally, a database represents a collection of data and the way it is organised. The interaction with a database is defined through a Database Management Software System (DBMS), a suite of pieces of software that allows access to all the information stored in the database. The DBMS contains methods for registering, updating and retrieving of information, as well as defining the structure of the information that will be stored.
All the functions supported by Database Management Software Systems can be grouped into four categories:

- Structure - the user will define the structure of the data that will be contained in the database, as well as all relationships between data
- Update - the user can insert, modify and delete data stored in the database
- Retrieval - the user can retrieve information from the database by specifying arbitrary criteria the data needs to match
- Access - the user can define access levels to the database. They can restrict the operations an user can use (as in read only access) or the access to certain parts of data.

The work in this report focuses on optimizing the retrieval of information from relational databases, with emphasis on Conjunctive Queries which heavily rely on joining multiple tables. Table joins can easily become a very expensive procedure which will slow down even basic queries if they are done in the trivial manner, which has an exponential worst-case time complexity.

M. Yannakakis discovered an algorithm that allows solving acyclic conjuctive queries in polynomial time in 1981 [21] and in 1998, Georg Gottlob, Nicola Leone and Francesco Scarcello published their results which allow 'weak cyclic queries', which are queries that have a bounded Hypertree Width, to be solved in polynomial time as well by Yannakakis algorithm using a method of decomposing the original query in a Hypertree Decomposition [5]. Since then, there has been a continuous research into more

advanced decompositions that would allow an even more generalized class of Conjunctive Queries to be solved in polynomial time. An amazing result of this research is the Fractional Hypertree Decomposition and in 2019, Wolfgang Fischl, Georg Gottlob and Reinghard Pichler published an algorithm for determining whether a Conjunctive Query of Fractional Hypertree Width k in polynomial time [20], by imposing certain restrictions to resulting decomposition.

In this report, I present my work of investigating the existing literature, pioneering the implementations of the Hypertree Decomposition [5] and Fractional Hypertree Decomposition [20] algorithms and empirically assess their utility outside the purely theoretical environment. The report will follow the process of implementing and altering the canonical algorithms in order to make them functional and to add the functionality of returning the decomposition of width k, if one exists, not just returning a boolean statement. At the same time, the report will present the results of testing these algorithms on various popular examples used in research papers and on uniformly sampled tests from Hyperbench [19].

Apart from the motivation presented so far, decomposition methods are useful in other areas such as: parallel query processing with MapReduce [3], bioinformatics [12], recommender system [15], combinatorial auctions [4] and virtually any problem Constraint Satisfaction Problem that has an underlying hypergraph of bounded width.

1.1 Goals of this project

The goals of this project were to:

- Study extensively the existing theoretical material in order to develop a solid knowledge base needed for implementing the algorithms.
- Bridge the gaps between the purely theoretical environment and the real world by creating efficient implementations of the Hypertree Decomposition algorithm [5] and Fractional Hypertree Decomposition algorithm [20].
- Assess the correctness of the decomposition algorithms specified above on relevant examples from research papers, as well as randomly selected samples from Hyperbench [19], which is a benchmark standard in the domain of hypergraph decompositions.

1.2 Personal Contributions

1.2.1 Theory

I had done extensive research to understand the theory behind the algorithms, their applicability and drawbacks, in order to be able to provide two of the first implementations of the Hypertree Decomposition and Fractional Hypertree Decomposition algorithms. The key notes that are necessary to read this project are presented in the Theoretical Foundation and Related Work chapters.

1.2.2 Implementation

I have created a complex code-base for providing all the auxiliary methods needed for the main Hypertree Decomposition [5] and Fractional Hypertree Decomposition [20] algorithms, such as efficiently storing a hypergraph, optimizing non-trivial mathematical operations that are used extensively in the main algorithms and created a modified version of the Depth-First Search algorithm to solve the sub-problem of finding [A]-Components that is not discussed in the original papers. Moreover, I pioneered the implementations of the main algorithms, Hypertree and Fractional Hypertree Decompositions, by simulating the essential alternating Turing Machine environment using Mutual Recursion [18].

1.2.3 Testing

I have compiled a solid test suite for all the auxiliary methods and classes, as well as provided the raw results of end-to-end tests on relevant inputs handpicked by myself from the existing literature and from Hyperbench.

1.3 Roadmap

- Chapter 2 introduces the reader to the theoretical foundation that is needed to understand both the origins of the problem the presented algorithms attempt to solve and how the algorithms actually work. Although no previous knowledge of graphs is required, it is considerably useful.
- Chapter 3 presents previous related work both theoretical and practical. It highlights a series of theoretical results that prove the relevance of these algorithms in the context of the academical problem of optimizing cyclic conjunctive queries. At the same time, it shows the results of previous experiments to introduce the importance of this process to the reader.
- Chapter 4 describes the details of implementation and the differences from the canonical algorithms presented as pseudocode in the original research paper that needed to be done to achieve practical relevance.
- Chapter 5 unveils the details of how different parts of the implementation have been tested, as well as presenting the results of a series of end-to-end tests for the main algorithms and the results drawn from them.
- Chapter 6 conveys the conclusion of this project as a result of both theoretical background and the practical approach of implementing and testing the algorithms.

Chapter 2

Theoretical Foundation

This chapter will present the key theoretical notions that revolve around the question "What do those algorithms do and why do we care?". While assuming the reader has no previous knowledge of graph theory, this chapter is not meant to be a lecture as it is far too succinct for that, but just as a brief method of refreshing knowledge or acquiring various missing terms.

2.1 Simple graphs

In mathematics, a graph $\mathcal{G}$ is a mathematical structure for modelling the relations between pairs. We note as $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ the graph $\mathcal{G}$ which is composed of the set of vertices (or nodes) $\mathcal{V}$, which are the individual instances we model, and the set of edges $\mathcal{E}$ which represent the relations between those instances. Although there are directed and undirected graphs, throughout the report, we will discuss only about undirected graphs, which we will simply refer to as graphs.

As we will later see, a key term of this project, and the main motivation of researching such algorithms, is represented by *cycles* in graphs. In the case of simple graphs, it is trivial to define and detect cycles, it will become far less obvious how to define a cycle when we move to hypergraphs. A *cycle* in a graph is any subset of at least two nodes which have the property that by traversing some of the edges incident to these nodes, we obtain a sequence of nodes in which just the first and the last one repeat. An acyclic graph that has the property that you can get from any node to all the others (it is connected) is called a tree.

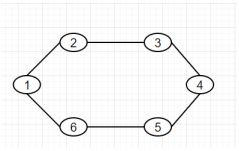

Figure 2.1: Example of a cyclic graph

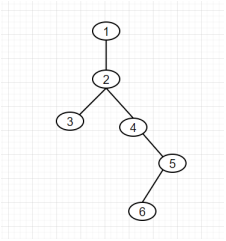

Figure 2.2: Example of an acyclic graph (a tree)

2.2 Hypergraphs

We can enhance the concept of edges within a graph by making them 'unite' more than just two nodes. Let $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ be a *hypergraph*, where $\mathcal{V}$ is the set of vertices and $\mathcal{E}$ is the set of hyperedges, where one hyperedge is a set of any size containing arbitrary vertices. It now becomes non-trivial to define cycles and there are actually multiple definitions of cyclicity within hypergraphs, we will focus on just α-cyclicity, for which we need to define the notion of an *ear*.

An *ear* in a hypergraph is any edge whose contained vertices can be split into two categories: vertices belonging to *just this hyperedge* and vertices that are also contained within at least one other hyperedge. If an hypergraph can be reduced to the empty hypergraph by a finite sequence of 'ear' removal operations, then it is acyclic. The GYO algorithm [2] exploits this process and verifies whether a hypergraph is cyclic through a finite sequence of ear removal operations.

A very important notion that will be a central piece in the algorithms implemented in this paper is that of a *component* in a hypergraph $\mathcal{H}$. Let A be a set of vertices of the hypergraph $A \in \mathcal{V}(\mathcal{H})$, two vertices, X and Y, of $\mathcal{H}$ are [A]-adjacent if there exists an edge e such that $\{X, Y\} \subseteq e - A$. We call X and Y [A]-connected if there is a path of vertices $(X, R_1, R_2, .., R_n, Y)$ in which each two pairs of nodes is [A]-adjacent. Let an [A]-component be a maximal subgraph of $\mathcal{H}$ in which any two nodes are [A]-connected.

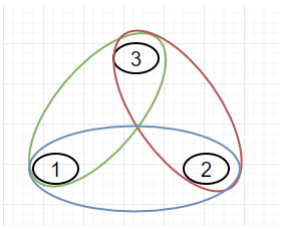

Figure 2.3: Example of a hypergraph with one cycle

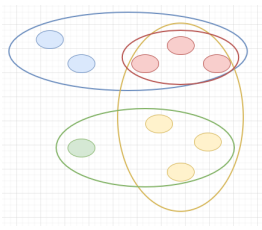

Figure 2.4: Example of an acyclic hypergraph

2.3 Conjunctive Queries

A database schema **R** is a set of relations, usually denoted by the symbols *R*, *T*, *S*, ... Each relation has an *arity* which denotes the number of attributes belonging to that relation. For simplicity, we will assume that attributes belonging to the same relation have the same domain they can take values from, denoted by **dom(R)**. A fact over relation *R* is an instance of *R*, an expression of the shape $R(a_1, a_2, ..., a_n)$, where n is the arity of *R*. A database instance **I** over a database schema **R** is a set of relational instances with $R \in \mathbf{R}$ and we denote with **adom(*R*)**, the set of constants present in the database instance.

Conjunctive queries are the simplest type of query that can be executed against a database instance and they are the ones the whole project is revolving around of. They are equivalent to a projection of the attributes of a relation *R* onto an arbitrary, not null set of attributes of *R*.

Throughout this paper, I will be using the Datalog notation for Conjunctive Queries as it is the standard in this field. Syntactically, a conjunctive query *q* (or simply CQ) is an expression of the form:

$$q(x_1, ..., x_k) : -R_1(y_1), ..., R_n(y_n)$$

where $n \geq 0$, $R_i \in R$ for every i = 1, . . . , n and q is a relation name. The expressions $x_1, x_2, .., x_k$ are called free variables, and contain either variables or constants. There are two syntactic requirements for writing a conjunctive query:

- The arity of the tuples y_k needs to match the arity of the relation R_k.
- Each variable x in $\langle x_1, x_2, ..., x_n \rangle$ needs to appear in the corresponding tuple y_k.

The expression $q(x_1, ..., x_k)$ is called the head of the query and it denotes the output of if, $R_1(y_1), R_2(y_2), ..., R_3(y_3)$ is the body of the query where each $R_k(y_k)$ is called an atom and multiple atoms can refer to the same relation. The head of the query can be left empty, without any variables, which means the Conjunctive Query is actually a Boolean query. The set of variables involved in the query *q* is denoted by **var(q)** .

The valuation v over a set of variables S is a total function that assigns to each variable in S a value from **dom(R)** . We can now formally define the semantics for CQs. Let **I** be a database instance over the schema **R**. Then, for the Conjunctive Query *q* the result *q*(I) of executing query *q* over the database instance I is:

$$q(I) = \{ \text{ v(head(q)) } | \text{ v is a valuation over } \mathbf{var}(q) \text{ such that } \forall \text{ i=1,...,n: } v(y_i) \in R_i^I \ \}$$

Example: We will execute query $q_1(x, y) : -R(x, y), T(y, z)$ over the relational instance: $I = \{R(a,a), R(a,b), R(b,c), R(c,a), S(b,c), S(b,b), T(c)\}$. There are two valuations over the set of variables { x,y,z } that satisfy the constraints of the CQ. The first one is v(x)=a, v(y)=b, v(z)=c since v((x,y)) = (a,b)$\in R^I$, v((y,z)) = (b,c) $\in S^I$. This valuation yields the resulting tuple (a,b). The other valuation is v(x)=a, v(y)=b, v(z)=b and the resulting tuple is still (a,b). Therefore, the final result of the CQ q over the specified

relational instance is $q_1(I) = \{(a,b)\}$.

2.3.1 Complexity of Conjunctive Queries

This subsection is dedicated to the topic that the algorithms implemented in this work are focusing on, which is optimizing the asymptotic time complexity of conjunctive queries.

It is useful to divide Conjunctive Queries into two categories based on their output: boolean CQ, whose output is either 'True' or 'False', and non-boolean, or full, CQ, whose output is a set containing a valuation of the variables in the head of the query with respect to the body of the query.

Let q be a non-boolean Conjunctive Query that we execute over the database instance I. We denote by $|q|$ the sum of the arities of the relations in the database instance, the number of variables in q, **var(q)** by k and by l, the number of atoms in q. Moreover, note that the following inequalities hold for any query: k,l $\leq |q|$. We use the trivial algorithm of computing all possible assignments of constants to variables to generate the result of q. This means that there is a maximum number $|\mathbf{adom(I)}|^k$ of possible valuations, which is polynomially bounded by $|\mathbf{adom}(I)| \leq |q| * |I|$, where $|I|$ is the cardinality of the database instance I . For each mapping, the algorithm checks if it is valid in linear time with respect to the number of atoms in q and we output it.

Therefore, the final asymptotic time complexity is

$$O(|I| + l * |\mathbf{adom}(I)^k|) = O(|q|^{k+1} + I^k).$$

From the previous equation one can observe that if we assume the Conjunctive Query q to be fixed and the database instance I to be the input, then the trivial algorithm of executing q against I has a polynomial time complexity. However, if we assume the database instance I to be fixed and the query q to be the input or having both of them as inputs, the time complexity of solving the Conjunctive Query becomes exponential.

2.3.2 Optimizing the Complexity of Conjunctive Queries

In the previous subsection, it was shown that the complexity of a non-boolean Conjunctive Query q when both the query q and the database instance I are given as input is NP-Complete. This subsection will show that there exists a polynomial time algorithm for evaluating a certain class of Conjunctive Queries.

Let us start with a simple example that will provide the reader with an insight on how the algorithm works intuitively. Let q be a Conjunctive Query which is boolean, for simplicity, the query q is defined as

$$q() : -R_1(x_1,x_2), R_2(x_2,x_3), ..., R_n(x_n,x_{n+1})$$

Moreover, let us assume for the sake of simplicity that every relation R_i has arity N. We say that the CQ q is a 'k-path query' as it does not present a cycle. Instead of trying all the possible valuations for solving the query, one can reduce the number of possibilities by applying the following algorithm. We first project $R_1(x_1,x_2)$ onto x_2

as x_1 is not needed in any join. Let $R_1^{'}(x_2)$ be the result of the projection, the next step is joining $R_1^{'}$ with $R_2(x_2,x_3)$ on x_2 and to project the result onto x_3, as we do not need x_2 for any other join operation. Let $R_2^{'}(x_3)$ be the result of the operations done so far, we will use this result in a similar manner until we exhaust all variables. The last operations done will join an arbitrary partial result $R_{n-1}^{'}(x_n)$ with $R_n^n(x_n,x_{n+1})$ and to project onto x_{n+1}. The facts in the resulting table $R_n^{'}$ are the results of executing the query q.

As we assumed that each relation R_i has arity N, it means that the total input size is N*n, where n is the number of relations in q, let us now analyze the time complexity of the presented algorithm. Each projection will take at most N steps, which means that in the worst case, the partial results $R_i^{'}$ will also have the size N. Joining two relations of size N can be done in linear time with respect to N by using a hash map, for example. Therefore, as both joining the relations and projecting onto one of the attributes have linear time complexity with regard to the size N of the relations, the final time complexity of executing the query is $O(n*N)$ which is linear with respect to the size of the input and therefore polynomial!

It needs to be noted that we assumed at the beginning that the Conjunctive Query q is boolean, and therefore the output is only 'True' or 'False', but for non-boolean queries, the output can be exponential with respect to the input in the worst case, thus making the design of an algorithm with polynomial asymptotic time impossible. The goal is then to design an algorithm that runs in polynomial time with respect to both the input and the output of a Conjunctive Query q. This subsection will present such algorithm and show that it can be applied only to the special subclass of queries which are acyclic.

We define the hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{E})$ of a Conjunctive Query q in the following manner. The set of vertices $\mathcal{V}$ is comprised of $\mathbf{var}(q)$, the set of all the variables in q. The set of hyperedges $\mathcal{E}$ is defined by the atoms of the query q.

A slightly altered version of the acyclic hypergraph $\mathcal{H}$ presented in Figure 2.4 corresponds to the following Conjunctive Query:

$$q() : -e_1(v_1,v_2,v_3), e_2(v_2,v_3), e_3(v_3,v_5,v_6), e_4(v_4)$$

The alteration comes from the fact that every vertex v_i needs to be covered by a hyperedge, which is equivalent with having the requirement that every attribute belongs to a relation in the query. Therefore, the vertex v_7 from the original hypergraph is discarded as it does not belong to any hyperedge.

Intuitively, a query is acyclic if its corresponding hypergraph is acyclic, but as already mentioned in the Hypergraphs subsection, this is not a trivial task and we need to resort to the sequential elimination of 'ears' from the hypergraph to assess if it contains cycles or not. The algorithm of reducing the hypergraph to the empty set by a finite sequence of 'ear' removal operations bears the name of the **GYO algorithm**.

In order to define an algorithm that will execute non-boolean queries in polynomial time, we need to define two more concepts that are true if the Conjunctive Query q is acyclic.

Let $\mathcal{F} = (\mathcal{V}, \mathcal{E})$ be a *join forest* for a Conjunctive Query q, where the set of vertices $\mathcal{V}$ is composed of the atoms of q with any atoms R and S that have at least one common attribute respecting the following requirements:

- R and S are in the same connected component of $\mathcal{F}$.
- all the common variables between R and S appear on a unique path from R to S.

If the *join forest* $\mathcal{F}$ of a CQ q contains a single connected component, then $\mathcal{F}$ is a *join tree*. A CQ q is acyclic if and only if it admits a *join tree*.

The last equivalent notion looks at reducing the time complexity of executing the query by limiting the amount of data that needs to be tested against the requirements of the query. Let a *dangling tuple* be a fact over the relation R that does not appear in the final valuation of a Conjunctive Query q.

For example, the Conjunctive Query q defined as following:

$$\text{q() :- } R\text{(x,y), } S\text{(y,z), } T\text{(z, w)}$$

Along with the database instance $I = \{R(a,b), S(b,c), T(c,d), R(a,b^{'}), S(b^{''},c^{'}), T(c^{'},d^{'})\}$. In this example, $R(a,b^{'})$, $S(b^{''},c^{'})$, $T(c^{'},d^{'})$ are all dangling tuples.

We can try removing dangling tuples by using semijoin operations. A semijoin between R and S is defined as a full join between R and S followed by a projection of the result on the attributes of R. If all *dangling tuples* can be removed by a finite sequence of semijoin operations, the sequence of operations is called a *full reducer*. An instance without any *dangling tuples* is globally consistent. A Conjunctive Query q admits a full reducer if and only if q is acyclic.

Using *join trees* and full reducers, the algorithm for running a non-boolean conjunctive query q over the database instance I in polynomial time with respect to the sizes of the input and output, which bears the name Yannakakis algorithm [21] , can be described as following: For input q which is a Conjunctive Query and I which is a database instance, the algorithm applies a full reducer to I, in order to obtain a globally consistent instance O; it computes the tree decomposition of the Conjunctive Query q and performs a post-order search in the tree, where at each step it performs a semi-join between the current node and its children.

Therefore, for any Conjunctive Query q and a database instace I of size N, evaluating q over N has the following time complexity:

- if q is boolean, then we can evaluate q in O(N).
- if q is full, then we can evaluate q in O(N + OUT), where OUT is the size of the output.

At this moment, a natural question should arise. Are acyclic queries the only subset of Conjunctive Queries that can be evaluated in polynomial time? The answer is yes, because the concepts Yannakakis algorithm uses are true only for acyclic queries, but in the next sections, tools for 'decomposing' a category of cyclic query into an acyclic one will be presented.

The formal motivation for this project can now be presented: Given a cyclic Conjunctive Query q, we want to determine if there exist any decomposition of q that will allow us to apply the Yannakakis algorithm on q. As it will be presented later, some methods of decomposition are more desirable than others. This work focuses on the implementation of Hypertree Decomposition and Fractional Hypertree Decomposition.

2.3.3 Decomposing Cyclic Queries

In the previous subsection it has been shown the Yannakakis algorithm, which can be used to evaluate a non-boolean Conjunctive Query over a database instance I in polynomial time. Even though the algorithm only works on acyclic queries, this subsection will present an example of a 'decomposition' of a cyclic query into a tree which can then be used as input for the Yannakakis algorithm.

Let us consider the following boolean Conjunctive Query q which detects a cycle of length k in the edges defined by the relation R, $q() : -R_1(x_1,x_2),R_2(x_2,x_3),R_3(x_3,x_4),...,$ $R_{n+1}(x_{n+1},x_1)$. Obviously, this query is cyclic, the reader can easily draw the hypergraph (which is just a graph in this case) corresponding to the query q to make sure of this.

Previously, the solution presented was to apply a series of projections and joins to get to the desired result. Let us try to apply the same process to this query, the first step would be to project $R_1(x_1,x_2)$ onto x_2 and then to join the result with $R_2(x_2,x_3)$, but if we do this, we lose the attribute x_1 which also appears in the last atom of the query and so it would make our query less restrictive and yield more results than it should. The solution is to keep x_1 around until the last join. So we join $R_1(x_1,x_2)$ with $R_2(x_2,x_3)$ and project the result onto (x_1,x_3). Let $R_1^{'}(x_1,x_3)$ be this partial result, the next step is to join it with $R_3(x_3,x_4)$ and project the result onto (x_1,x_4). One needs to notice that this time, the time complexity of the worst case for both of the steps is N^2 instead of just N and the last result will be (x_1,x_{n+1}) which can have size at most N^2. Therefore, we can evaluate the cyclic Conjunctive Query q over the database instance I in $O(n*N^2)$ time complexity, which is still polynomial.

One can view this algorithm as creating 'decomposition' of the initial query in order to reduce the unnecessary work that is done. From an intuitive point of view, one could view the algorithm as transforming the initial Conjunctive Query q into a new query $q^{'} : -S_1(x_1,x_2),S_2(x_1,x_2,x_3),S_3(x_1,x_3,x_4),...,S_n(x_1,x_n,x_{n+1})$ which is equivalent to the first one, but is *acyclic*!
Therefore, we can apply Yannakakis algorithm on the new query.

2.4 Edge covers

This section formalises the process through which Conjunctive Query decompositions can be used to optimize the time complexity of evaluating a query by introducing the notion of Edge Cover.

Let $\mathcal{H} = (\mathcal{E},\mathcal{V})$ be a hypergraph and let $\lambda : \mathcal{E} \to \{0,1\}$ and $\gamma : \mathcal{E} \to [0,1]$ be two functions defined on the set of edges of $\mathcal{H}$. We define the set $B(\xi)$ of covered vertices,

where $\xi \in \{\lambda, \gamma\}$, as $B(\xi) = \{v \in \mathcal{V}(\mathcal{H}) | \sum_{e \in \mathcal{E}(\mathcal{H}), v \in e} \xi(e) \geq 1\}$. Intuitively, the set of covered vertices of $\mathcal{H}$ is comprised of the set of vertices whose sum of the weights belonging to the hyperedges that contain them is above 1. The weight of such a cover is the sum of its weights: weight(ξ) = $\sum_{e \in \mathcal{E}(\mathcal{H})} \xi(e)$

Although both λ and γ are edge covers, throughout the rest of this work, λ will be referred to as the Integral cover of edges and γ as the Fractional cover of edges and will be addressed separately as they are the foundation of two different algorithms.

The goal is to find the edge cover with minimal weight that covers all the nodes of the hypergraph $\mathcal{H}$.

2.4.1 Integral Edge Cover

Intuitively, an integral edge cover assigns to each each edge a natural number, namely either 1 or 0. To respect the notation used in [5], one can view the integral edge cover λ as a subset of edges of the hypergraph $\lambda \subseteq \mathcal{E}(\mathcal{H})$, comprised of all the edges for which $\lambda(e) = 1$. This will ease the understanding of the Hypertree Decomposition algorithm. However, this work respects the extended notations used in this work [20]. The edge cover number $\rho(\mathcal{H})$ is the is the minimum weight of all edge covers of $\mathcal{H}$.

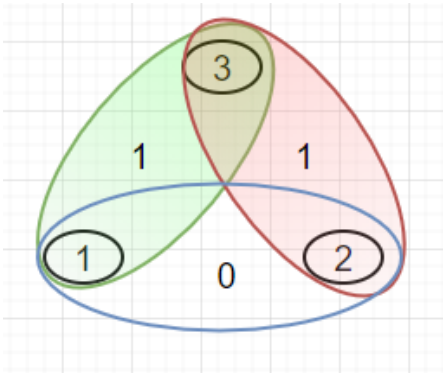

Figure 2.5: Example of an integral cover

2.4.2 Fractional Edge Cover

A fractional edge cover assigns to each edge a rational number between 0 and 1 included. We define the goal of a Linear Program (LP) as being:

$$\text{minimize: } \sum_{e \in \mathcal{E}(\mathcal{H})} \gamma(e)$$

We also define a restriction for our LP as being:

$$\sum_{e \in \mathcal{E}(\mathcal{H}), v \in e} \gamma(e) \geq 1, \forall v \in \mathcal{V}(\mathcal{H})$$

In intuitive terms, the goal of the linear program is to assign to each edge a weight between 0 and 1 such that the sum of weights over each vertex is at least 1. This goal is equivalent with finding a minimal edge cover for the hypertree $\mathcal{H}$. Let $\rho'(\mathcal{H})$ be the the minimum edge cover of hypergraph H.

Obviously, $\rho(\mathcal{H}) \geq \rho_{\prime}(\mathcal{H})$ for any hypergraph $\mathcal{H}$ and in fact, $\rho'(\mathcal{H})$ can be much smaller, which makes fractional covers more a more appealing method for decomposing queries as they have more power to generalize to a bigger set of Conjunctive queries as it will be presented in the next chapter.

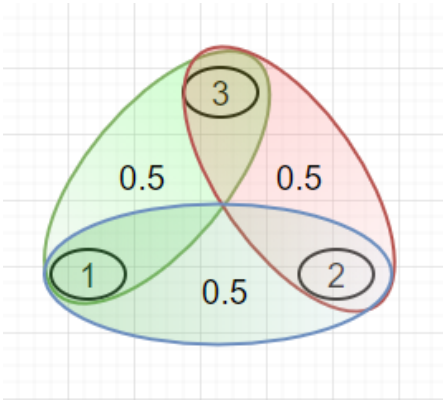

Figure 2.6: Example of fractional cover

Chapter 3

Related Work

This chapter will present the desired properties of the ideal decomposition method as well as describe the main attempts to generalize the property of acyclic queries of being evaluated in polynomial time.

Finding more powerful ways to generalize query acyclicity is essential in discovering methods for solving problems which were initially believed to be part of NP in polynomial time. This is a direct result of Courcelle's theorem: if a graph can be expressed in *Extended Monadic Second Order Logic*, then for every upper-bounded tree width $w \geq 1$, there exists an algorithm for testing that property of the graph in linear time. *Extended Monadic Second Order Logic* contains:

- first order logic
- adjacency and incidence relations
- quantification over sets of vertices and sets of edges

It turns out that many problems are polynomially solvable for bounded tree-width, for example: vertex colouring, edge colouring, Hamiltonian cycle, maximum clique, vertex disjoint path, constraint satisfaction problem.

Therefore, research in the domain of query decompositions has the potential of reshaping our theoretical knowledge of the time complexity of certain problems, but so far, little practical experiments have been made to assess the utility of such algorithms. This work addresses the missing bits in the pseudocodes presented in [5] and [20] as well as revealing the practical implementations of such algorithms and their behaviour on different test suites.

3.1 How to Generalize Query Acyclicity?

Any good decomposition method must respect three criteria:

- *Generalization of acyclicity*, which means that all queries with width w ≥ 1 will include all the acyclic ones.

- *Tractable recognizability*, which means that any decomposition of width w can be computed efficiently.
- *Tractable query-answering*, which means that using the obtained decomposition, one can efficiently compute the solution of a Conjunctive Query q.

These three criteria will be used to assess different methods of decomposing a hypergraph that have been studied in the past.

3.2 Treewidth

For a hypergraph $\mathcal{H}$, we define a tree decomposition as a tuple $(\mathcal{T}, \mathcal{B})$, where $\mathcal{T}$ is a tree and $\mathcal{B}_{v,v\in T}$ is a subset of vertices of $\mathcal{H}$ called the bag of node v, such that for each edge $e \in \mathcal{E}(\mathcal{H})$, the vertices of e are included in one of the bags of the tree decomposition $\mathcal{T}$ and for every vertex v $\in \mathcal{V}(\mathcal{H})$, the set of nodes $\{t \in \mathcal{V}(\mathcal{T}) | v \in \mathcal{B}_t\}$, which is the set of nodes that contain v in their bags, is connected in $\mathcal{T}$.
The tree width of a tree decomposition $(\mathcal{T}, \mathcal{B})$ is the cardinality of the maximal length bag in $\mathcal{B}$. The tree width of a Conjunctive Query q is the minimal tree width over all the possible decompositions of q.

Referring back to the criteria defined in the previous section, tree width satisfies the following:

- Optimal tree decompositions of fixed width k can be computed efficiently [10].
- Queries of bounded treewidth k can be evaluated efficiently [1].

However, treewidth breaks the first criterion, as one can see in the following example:

Let a Conjunctive Query q be defined as $q() : -R(x_1, x_2, x_3, .., x_n)$

Obviously, the tree width of this query is n because the can be only one node in the decomposition, which contains all n attributes, but the query q is acyclic as it contains only one node. So treewidth decomposition fails to properly capture the cases in which the CQ q is acyclic.

There are several other invariants which are equivalent or within a fixed constant from treewidth that have been studied in the past. They include: bramble number [13], branch number [11] and the linkedness [13].

Let us take a simple hypergraph in figure 3.1 to show examples of decompositions.

3.3 Primal graphs

Let a Primal Graph (or a Gaifman Graph) $\mathcal{G} = (\mathcal{V}(\mathcal{H}), \mathcal{E}^{'}))$, with $\mathcal{E}^{'} = \{\{x, y\} | x \neq y, \exists e \in \mathcal{E}(\mathcal{H})$ such that $\{x, y\} \in e\}$ of a hypergraph $\mathcal{H}$ be the graph whose set of vertices is comprised of the set of vertices of $\mathcal{H}$ where any two vertices are connected in the Primal Graph $\mathcal{G}$ if they belong to the same hyperedge e of $\mathcal{H}$.

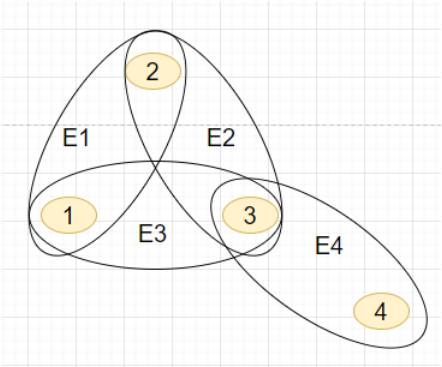

Figure 3.1: Example of a hypergraph with a cycle

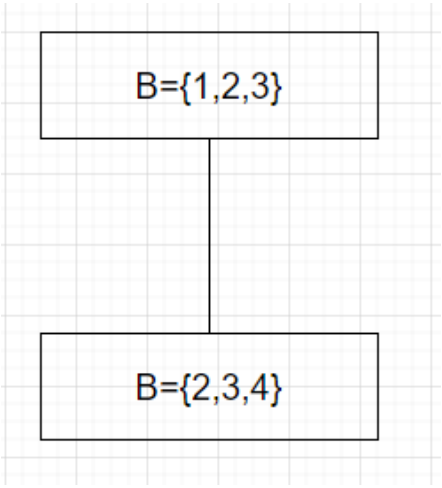

Figure 3.2: Example of tree decomposition for the hypergraph in figure 3.1 with tree width = 3

Primal Graphs enjoy a special property which makes them appealing: for any treewidth w, there is a linear-time algorithm for solving a Constraint Satisfaction Problem whose Primal Graph has treewidth at most w. Moreover, their simplicity compared to hypergraphs was an incentive to study the properties of Gaifman graphs.

However, hypergraphs provide much more precision when calculating the complexity analysis of a conjunctive query q. For example, let us consider the queries q_1 and q_2 defined as

$$q_1 : -R(x_1,x_2,x_3,..,x_n) \text{ and } q_2 : -R_1(x_1,x_2),R_2(x_2,x_3),R_3(x_3,x_4),...,R_{n-1}(x_{n-1},x_n)$$

The Conjunctive Queries q_1 and q_2 have the same Primal graphs, but if we consider the hypergraphs, we will see in the following chapters that q_1 has in fact a bounded width and it is an easy example, as opposed to q_2 which is a hard example.

This is because if an edge covers every vertex, we have at most $|I|$ possible solutions, but if the vertices are covered by k edges, we have at most $|I|^k$ possible solutions.

3.4 Generalized Hypertree Decomposition

For a tree $\mathcal{T} = (\mathcal{V}, \mathcal{E})$ and a node v $\in \mathcal{V}$, let $\mathcal{T}_v$ be the subtree of $\mathcal{T}$ rooted in v, that is the subset of nodes in $\mathcal{T}$ that are reachable from v.

Let a Generalized Hypertree Decomposition of a Conjunctive Query q be a tuple $(\mathcal{T}, \mathcal{B}, \mathcal{C})$, where $(\mathcal{T}, \mathcal{B})$ is the tree decomposition of q. The set $(\mathcal{C}_t)_{t \in \mathcal{V}(\mathcal{T})}$ is an arbitrary set of edges of the hypergraph $\mathcal{H}$ corresponding to q, called the guards of the nodes in the decomposition of q. For $\mathcal{C}$ to be the set of guards of the decomposition, it has to respect the following criterion: for any node t in $\mathcal{V}(\mathcal{T})$, we have $\mathcal{B}_t \subseteq \cup\mathcal{C}_t$ where $\cup\mathcal{C}_t$ defines the set of vertices $\{v \in \mathcal{V}(\mathcal{H}) | \exists e \in \mathcal{E}(\mathcal{H}) : v \in e\}$.

Intuitively, this just means that the set of vertices defined by the bag of t $\mathcal{B}_t$ is included in the set of vertices defined by the union of the edges in the set of guards of t $\cup\mathcal{C}_t$.

The width of the decomposition $(\mathcal{T}, \mathcal{B}, \mathcal{C})$ is given by $max\{|\mathcal{C}_t| t \in \mathcal{V}(\mathcal{T})\}$, which is the biggest cardinality over all the sets of guards of the decomposition. The width of a Conjunctive Query q is the minimal width over all the possible Generalized Hypertree Decompositions of q.

Assessing GHDs by the criteria presented at the beginning of the chapter, we notice the following:

- all queries with width w ≥ 1 will include all the acyclic ones. Therefore GHDs respect the first criterion which is Generalization of Acyclicity.
- queries of bounded ghw w can be evaluated efficiently. Therefore GHDs respect the third criterion which is Tractable-Query Answering.

Unfortunately, it has been proven that checking whether ghw(Q)=3 is NP-complete [7], therefore Generalized Hypertree Decompositions break the second criterion, Tractable Recognizability, as they cannot be computed in efficient time.

However, we can add an 'artificial' restriction to Generalized Hypertree Decomposition that will not preserve the ability to respect the first two criteria and will also allow the new decomposition to be found in efficient time.

3.5 Hypertree Decomposition

Let a Hypertree Decomposition of a Conjunctive Query q be a Generalized Hypertree Decomposition $(\mathcal{T}, \mathcal{B}, \mathcal{C})$ which also respects the following special requirement: for each node v in $\mathcal{V}(\mathcal{T})$, $\mathcal{V}(\mathcal{T}_v) \cap \mathcal{C}_t \subseteq \mathcal{B}_t$. This intuitively means that if a variable k disappears from $\mathcal{T}$ at a certain node, the same variable will not appear anywhere else at lower levels of the tree. The Hypertree Width of a decomposition is equal to the maximal cardinality over all sets of guards $\mathcal{C}$ of the decomposition. The Hypertree Width of q is the minimal hw over all possible decompositions of q.

So far, Hypertree Decomposition is the only known method of decomposing a Conjunctive Query q that respects all three criteria of an ideal decomposition as it has

the same properties as Generalized Hypertree Decomposition, but the special condition allows it to be computed in polynomial time as presented in this paper [5]. The downside is that the time complexity although is polynomial, the constant is high; deciding whether a Conjunctive Query q has a hypertree width k has an asymptotic time complexity $O(|V| * |q|)^{2*k}$.

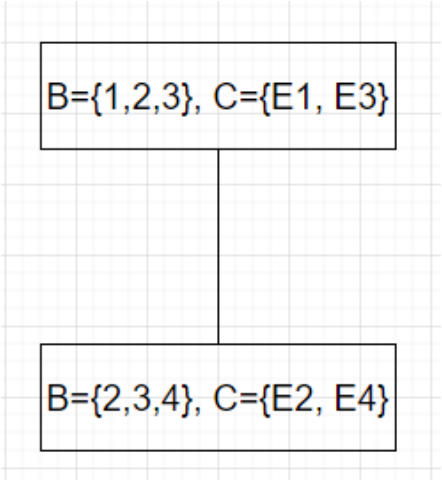

Figure 3.3: Example of Hypertree Decomposition for the hypergraph in figure 3.1 with hypertree width = 2

In practice, ghw and hw do not differ much, the relation between the two metrics, which is presented in this paper [6], is the following:

$$\text{For any Conjunctive Query } q,\ ghw(q) \leq hw(q) \leq 3 * ghw(q) + 1$$

3.6 Fractional Hypertree Decomposition

A Fractional Hypertree Decomposition of a hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{E})$ is a tuple $(\mathcal{T}, \mathcal{B}, \gamma)$ which respects the following requirements:

- for each edge $e \in \mathcal{E}(\mathcal{H})$, the vertices of e are included in one of the bags of the tree decomposition $\mathcal{T}$.
- for every vertex v $\in \mathcal{V}(\mathcal{H})$, the set of nodes $\{t \in \mathcal{V}(\mathcal{T}) | v \in \mathcal{B}_t\}$, which is the set of nodes that contain v in their bags, is connected in $\mathcal{T}$.
- for every node t $\in \mathcal{V}(\mathcal{T})$, γ_t is a function such that $\mathcal{B}_t \subseteq \mathcal{B}(\gamma_t)$, where $\mathcal{B}(\gamma_t)$ denotes the set of nodes that are covered by a fractional edge cover function γ, $\mathcal{B}(\gamma_t) = \{v \in \mathcal{E}(\mathcal{H}) | \sum_{e\ in \mathcal{E}(\mathcal{H}): v \in e} \gamma(e) \geq 1\}$.

The Fractional Hypertree Width of a decomposition is the weight of the fractional cover γ. The Fractional Hypertree Width of a query q is the minimal fhw over all the possible Fractional Hypertree Decompositions of q.

The relation between all the decomposition methods presented so far is the following:

$$\text{For any Conjunctive Query } q,\ fhw(q) \leq ghw(q) \leq hw(q) \leq th(q) + 1.$$

Although it is demonstrated in this paper [20] that the problem of checking if a Conjunctive Query q has a Fractional Hypertree Width of k for $k \geq 2$ is NP-Complete, the authors also show that for queries that benefit from a certain property (queries that have a bounded intersection of hyperedges) the problem is tractable and can be computed efficiently.

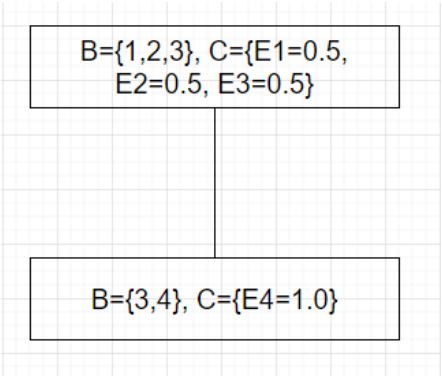

Figure 3.4: Example of Fractional Hypertree Decomposition for the hypergraph in figure 3.1 with fractional hypertree width = 1.5

Chapter 4

Implementation

This chapter presents the two algorithms implemented in this work by detailing the canonical pseudo-code, the way they intuitively work and the alterations that had to be done to be able to make them work. The language used was Java and therefore, certain particularities of the language will be present in the following sections.

4.1 Hypergraph Representation

The first piece of work implemented was the object that allows the storing of a hypergraph such that all the needed operations will be done efficiently. Although in works such as [5], [20], [9] and others, a hypergraph is defined as a tuple containing a set of vertices and a set of edges, which is equivalent to a set of sets, I opted to represent the hypergraph as an 'ArrayList' of edges where each element is a 'HashMap' denoting the id of the vertices belonging to that edge and their weight. The reason I chose this way of storing the hypergraph might be clearer after reading the implementation for Fractional Hypertree Decomposition.

Apart from storing just the edges of the hypergraph, I chose to also implement a complementing 'HashMap' that maps each vertex v to a set of edges it belongs to. This complementing structure proved to be very useful especially when computing joins of arbitrary sets of edges, as well as computing sub-hypergraphs, which is a quintessential step in finding a Fractional Hypertree Decomposition. As in the papers on which my implementation is based the authors use mathematical operations such as intersection of two sets which are trivial mathematically, but can sum up to a very high cost if not implemented correctly, optimization details such as the 'HashMap' from vertices to edges proved to be useful given the amount of repetition of the operation.

The hypergraph class makes use of Java Generics to generalize the type of ID used for vertices. The only requirement is that the type used extends the class 'Comparable' which is essential for the 'HashMaps' as they use the ID's as keys. This allows users to use any comparable type of data as keys for vertices seamlessly. The class also provides a helper function to efficiently compute the intersection between an edge and a set of edges, making use of the complementary structure that maps vertices to edges

for determining common vertices.

4.2 [A]-Components

Both algorithms revolve around [A]-Components to find a decomposition that has a weight $w \leq k$, where k is given as an input to the algorithm along with the hypergraph. Although in both pseudocodes, [A]-Components are treated as just bags of vertices, this proved to be unattainable in practice because the same vertices can be covered by multiple subsets of edges which will yield different results. Therefore, in practice, both the set of vertices in an [A]-Component and the set of edges that cover those vertices are essential for the algorithms.*This is the first alteration of the canonical algorithms that was needed in order to implement them.*

Therefore, the implementation makes use of a simple 'Component' class which just contains a 'HashSet' for the vertices of the component and a 'HashSet' for the covering edges.

4.3 Hypertree Decomposition

The first algorithm implemented was Hypertree Decomposition. The implementation is based on the theory and pseudocode presented in this paper [5] which represents the foundation in research on this topic. I will firstly present the canonical algorithm and an intuitive description of how it works, then I will present my ways of dealing with the missing bits in the pseudocode and the alterations suffered by the original version.

ALTERNATING ALGORITHM k-decomp
Input: A non-empty Query Q.
Result: "Accept", if Q has k-bounded hypertree width; "Reject", otherwise.

Procedure *k-decomposable(C_R: SetOfVariables, R: SetOfAtoms)*
begin
1) **Guess** a set $S \subseteq atoms(Q)$ of k elements at most;
2) **Check** that all the following conditions hold:
   2.a) $\forall P \in atoms(C_R),\ (var(P) \cap var(R)) \subseteq var(S)$ and
   2.b) $var(S) \cap C_R \neq \emptyset$
3) **If** the check above fails **Then Halt** and **Reject**; **Else**
   Let $\mathcal{C} := \{C \subseteq var(Q) \mid C \text{ is a } [var(S)]\text{-component and } C \subseteq C_R\}$;
4) **If, for each** $C \in \mathcal{C}$, *k-decomposable*(C, S)
   Then Accept
   Else Reject
end;

begin*(* MAIN *)*
   Accept if *k-decomposable*$(var(Q), \emptyset)$
end.

Figure 4.1: Hypertree Decomposition algorithm pseudocode presented in [5]

On an intuitive level, the authors of this paper first described the Normal Form of a Hypertree Decomposition which is a more strict version of the normal decomposition, equivalent with a 'bag-maximal hypertree decomposition' described in this paper [20].

A Hypertree Decomposition $(\mathcal{T}, \mathcal{B}, \mathcal{C})$ of a Conjunctive Query q is in normal form if the following requirements are respected for any node r $\in \mathcal{V}(\mathcal{T})$ and each node s that is child of r:

- there is only one [$\mathcal{B}_r$]-component, C_r, such that all the vertices that appear in the bags of the nodes belonging to the subtree rooted in s, $\mathcal{T}_s$ are included in the set defined by the reunion the of the vertices in C_r and the intersection of the bags belonging to r and s, $\mathcal{B}(\mathcal{T}_s) = C_r \cup \mathcal{B}_s \cap \mathcal{B}_r$.
- there are no common vertices between the bag of s and the component from the previous requirement, $\mathcal{B}_s \cap [\mathcal{C}_r] = \emptyset$.
- the intersection between the set of vertices defined by the guards of s and the vertices in the bag of r is included in the set of vertices defined by the bag of s, $\cup \mathcal{C}_s \cap \mathcal{B}_r \subseteq \mathcal{B}_s$.

These requirements allow us to induce a deterministic component into the algorithm instead of counting on only random walks to determine the hypertree decomposition that has a hypertree width smaller than k, thus optimizing the time efficiency of the algorithm.

Let us delimit the pseudocode presented in figure 4.1 into four sections which will aid the analysis of the whole implementation. The step marked with '1)' in the pseudocode is the component that marks the random walk in the solution space, the algorithm 'guesses' guards for each node of the hypertree and then checks it they respect the requirements. The second step actually determines if the current assignment of guards respect the needed properties for the solution to be a Hypertree Decomposition in normal form. If the current partial decomposition respects all the requirements, the third step of the algorithm goes over all the possible components that represent the search space for the subproblems. The last step is checking if the current decomposition can lead to any complete one.

4.3.1 Personal Contributions to the HD algorithm

There are several ambiguities in the algorithm presented in this paper [5], I will address this ambiguities in this subsection, along with presenting my personal solutions which have been introduced in the implementation of the algorithm.

The authors of the paper described the canonical algorithm as "a high level description of an alternating algorithm, to be run on an alternating Turing machine (ATM)". An alternating Turing machine is a non-deterministic finite state machine that alternates between existential states (states that are accepted if at least one transformation leads to an accepting state) which correspond to the NP class and universal states (states that are accepted if and only if all transformations lead to an accepting state) which correspond to the co-NP class.

The alternating nature of the algorithm stems from the fact that all the possible components generated in step 3 need to be valid for a decomposition to be valid, but only one

guess from step 1 needs to be valid for the algorithm to have found a correct decomposition. Thus, the first problem I encountered was the simulation of the alternating behaviour necessary for the algorithm. I successfully simulated the behaviour of an alternating Turing machine using a programming paradigm known as Mutual Recursion [18], which consists of two functions which call each other to form a recursive loop. This allows the logic that checks if the current assignment of the variable S respects the needed requirements to be separated from the logic of assigning values to S. Therefore, if any check from the steps 2) or 3) fails, the recursion tree for that partial solution can be terminated and the algorithm can still continue to test every other possible assignment of S until it exhausts the options. A high level description of the implementation is presented in figures 4.2 and 4.3.

Algorithm 1: Hypertree Decomposition(Existential part)

Input: a hypergraph G, a natural number k representing the hypertree width, an [A]-Component object, a Set representing the guards of the parent node and a set representing the vertices covered by the guards

Output: a boolean that specifies if a hypertree decomposition of width at most k has been found and the decomposition

```
for S in all the possible subsets of at most k hyperedges from the component do
    if Hypertree Decomposition (Universal part) == true then
        return true;
    else
        S = getNextS();
    end
end
```

Figure 4.2: The existential part of my implementation of the Hypertree Decomposition Algorithm

The next ambiguity refers to the 'guess' of the subset S in step 1. If one let's the 'guess' to be a random uniform distribution, there is a risk of testing the same partial solution multiple times and thus wasting resources. Obviously, one could use a structure like a Set to keep track of previous tried possible solutions, but this would require more memory and would still consume time. The solution implemented by me is to use a deterministic approach. I created a class 'GetSets' which given a hypergraph, generates all the possible combinations of sets of at most k hyperedges. The class computes the combinations in an efficient manner by keeping track of the last combination it returned and in order to optimize the memory consumption of the total algorithm, the class only returns one possible combination of hyperedges per call, therefore, multiple calls need to be done to exhaust them. In this way, the program will not have a massive overhead from storing even the sets of hyperedges that will not be used.

Moreover, the fact that I save the [A]-Components as both a set of vertices and a set of edges that cover them using my custom class allowed me to optimise the runtime of the original algorithm. In the canonical pseudocode, the authors described the first step as a guess of a set of arbitrary hyperedges of cardinality at most k. A non-trivial observation can be made that the same hyperedge will be used multiple times if and only if some of the vertices it covers are also used in another node of the tree decomposition. Therefore, instead of guessing arbitrary hyperedges from the set $\mathcal{E}(\mathcal{H})$, we

Algorithm 1: Hypertree Decomposition(Universal part)

Input: a hypergraph G, a natural number k representing the hypertree width, an [A]-Component object $Comp_r$, a set $Guards_r$ representing the guards of the parent node, a set $Vertices_r$ representing the vertices covered by the guards and the set S of guessed guards

. **Output:** a boolean that specifies if a hypertree decomposition of width at most k has been found and the decomposition

.

1. Check if the following requirements hold:
   $\forall C \in Comp_r$, var(C) $\cap Vertices_r \subseteq var(S)$
   var(S) $\cap C \neq \varnothing$
2. Get all the possible components for the children of this node
   listOfComponents $\leftarrow$ getPossibleComponents($Comp_r, var(S)$)
3. Compute the bag of vertices for this node according to the following relation:
   $\mathcal{B}(currentNode) = C_r$ if this is the root of the tree or
   $\mathcal{B}(currentNode) = var(S) \cap \mathcal{B}_r \cup Comp_r$
4. Iterate through all the possible components and check if they all lead to a valid decomposition

```
for C_i in listOfComponents do
    if Hypertree Decomposition (Existential part)(G, C_i, S, var(S) ) == false then
        return false;
    end
end
return true ;
```

Figure 4.3: The universal part of my implementation of the Hypertree Decomposition Algorithm

can use only the set of hyperedges that cover the vertices in the component. While this optimization does not improve the asymptotic time complexity, it improves the practical run-time of the algorithm by reducing the number of possible subsets.

Another alteration of the canonical algorithm, which I have deemed to be necessary through empirical testing, deals with the checks at the second step in the Hypertree Decomposition (universal part) algorithm. The canonical algorithm only checks whether var(C) $\cap Vertices_r \subseteq var(S)$, where C is a hyperedge in the $[C_r] - Component$, and where r is the parent of the current node. Mathematically, the intersection of two sets can yield the null set as a result, which is trivially included in each arbitrary set, thus satisfying the requirement established by the authors. Through empirical tests, I discovered that this constraint is not enough for the algorithm to work properly as it might ignore cycles in a hypergraph. Let us take an example to illustrate this:

As we can see in the figure 4.4, the algorithm ignores the cycle of the hypergraph and selects the hyperedges that have a blue hue. This is because the algorithm first selects the top-left hyperedge, then at the next stage, it selects the bottom-right one, which respects the conditions in the original algorithm because var(C) $\cap Vertices_r$ yields the null set which verifies the requirement. Intuitively, this means that the algorithm allows nodes adjacent in the tree decomposition to not have any vertices of the hypergraph in common, even more, it even means that there are no vertices covered by the guards of the nodes which are common. This can lead to unwanted behaviour such as the one described in this example. My solution is simple, I introduced another requirement in the original check that does not allow the intersection var(C) $\cap Vertices_r$ to yield the null set.

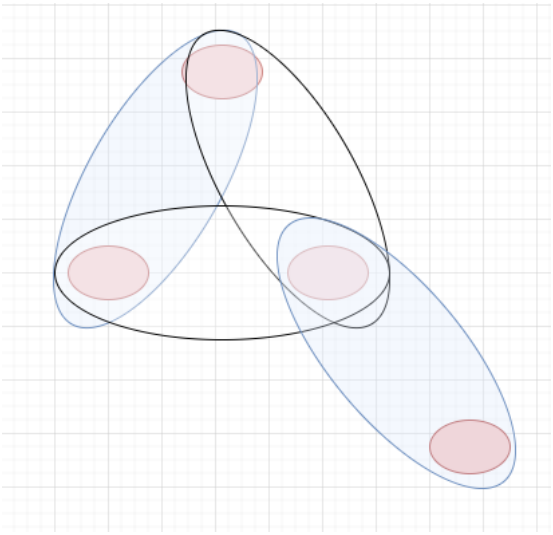

Figure 4.4: Example of hypergraph where the canonical check at part 2 fails

The next problem I will address refers to step 3) in the pseudocode presented at figure 4.1. In the original paper, the authors do not detail any algorithm for finding a [C]-Component, where C is an arbitrary set of vertices. As this is a vital part of the algorithm which ensures that the final hypertree decomposition will have the required properties, I needed to come up with an algorithm for computing a [C]-Component in an efficient manner as the goal of implementing these algorithms is to assess their practical utility. I created an adaption of the Depth-First Search algorithm [15] in order to provide the correct functionality. A high-level presentation of my modified version of DFS is presented at the figures 5.9 and 4.7.

The algorithm for the sub-task of finding all [C]-Components is implemented using two methods, 'getPossibleComponents' which deals with the case in which we have multiple components that are not connected with each other and 'DFS' which finds the maximal [C]-connected component. The intuition behind the algorithm is simple, but let us take an example so it is easier to follow:

In the example 4.5 there is presented a hypergraph along with its three [E1]-Components coloured by green, red and purple. The green component is defined by E5, E6, E7 and the red one by E2 and E3 and the purple one by E4. At the invocation of the function 'getPossibleComponents', the algorithm loops through all vertices and if they haven't been visited and they are not part of E1, it invokes the DFS function on that vertex. Let us assume that red will be explored first, the algorithm performs a DFS search by visiting all vertices that are contained by the same hyperedge as the current one and marks them as visited. The algorithm will thus explore all vertices belonging to the hyperedges E2 and E3 and that do not belong to E1 as well.

Once all nodes have been explored, DFS finishes and returns to 'getPossibleComponents' where the current explored component will be saved. 'getPossibleComponents' will then loop through all vertices and find the first vertex that has not been explored and does not belong to E1. Let us assume it will find the only node that belongs exclusively to E4, the algorithm will realize that there is only one node that does not

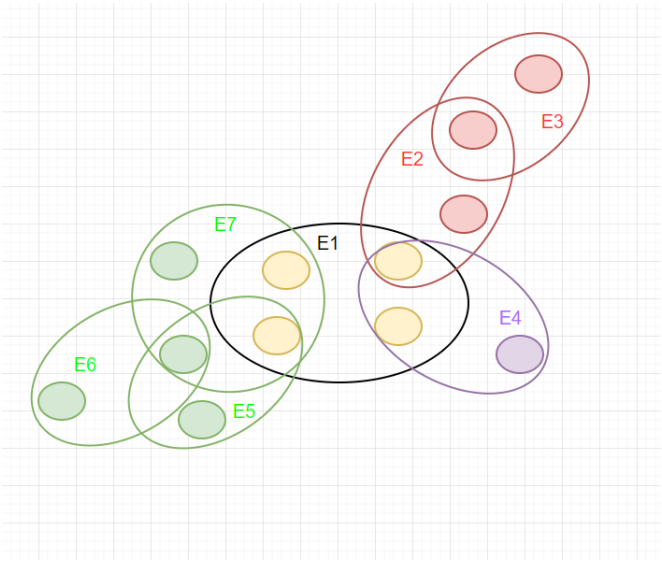

Figure 4.5: Example of components detected by the modified DFS

belong to E1 and will form the purple component. Finally, the last component will be explored, which is the green one, one can see that even though E7 has 2 vertices from E1, it also allows two vertices to be [E1]-adjacent and therefore respects the condition of being included in the component. The final component will be formed of E5, E6, E7.

Although the original description of an [A]-Component in both [20] and [5] states that a component is the maximal subgraph formed by pairs of vertices (X,Y) which are [A]-Adjacent, an alteration has been made during the implementation of the 'getPossibleComponents' function as in practice, it proved necessary to allow components to be determined even by only one vertex, such as the purple component in example 4.5.

Algorithm 3: getPossibleComponents

Input: a hypergraph G, a set S which represents the vertices used in the current assignment of the guards

. **Output:** a list of [S]-Components

.

allComponents ← {} **for** *vertex in* $\mathcal{V}(\mathcal{T})$ **do**
  component ← new Component(); **if** *vertex has not been used in a component already and is not part of S* **then**
    DFS(vertex, S, component) allComponents.add(component)
  end
end
return allComponents

Figure 4.6: The pseudocode for the getPossibleComponents method

Proof: As already stated in the definition of [C]-Components, a [C]-Component is a maximal subgraph where every two vertices are [C]-connected. I will now prove that DFS will find the maximal subgraphs using a proof by counterexample. Let us assume that in the example presented, there is another vertex which can belong to the

```
Algorithm 4: DFS
Input: a vertex currentVertex, a set S which represents the vertices used in the
       current assignment of the guards, currentComponent which is an empty
       Component instance which is an output parameter
.
if currentVertex has been explored already then
    return
end
for hyperedge in the set of hyperedges that cover currentVertex do
    for vertex in hyperedge do
        currentComponent.add(hyperedge)
        if vertex is not part of S then
            currentComponent.add(vertex) DFS(vertex, S, currentComponent)
        end
    end
end
return allComponents
```

Figure 4.7: The pseudocode for the DFS method

red component and which will not be explored by the DFS algorithm. Let us name this new vertex V_k, by definition, if V_k can be added to the red component it means that all the vertices in the red component are [C]-connected with V_k. Therefore, there exists at least one edge E_k which contains V_k and another vertex $V_u$that is already in the red component, such that V_k and V_u are [E1]-adjacent. But this means that when the DFS algorithm will explore all the vertices that are in all the hyperedges which contain V_u, the DFS search will automatically also explore V_k which contradicts the initial assumption.

Therefore, the modified DFS algorithm will return maximal subgraphs which are [C]-connected. As the algorithm will check all the vertices which are adjacent with the current one by going through all the hyperedges which contain the current vertex, the worst asymptotic time complexity is still O(N * M) where N is the number of vertices and M is the number of hyperedges, thus it will not influence the general complexity of the algorithm and will add just a light overhead in the run-time, therefore solving the problem in an efficient manner.

It is worth mentioning that another alteration has been made to the original algorithm which is represented by adding additional parameters to the function, 2 of which will contain the final hypertree decomposition of width k, if there is one.

4.4 Fractional Hypertree Decomposition

The second algorithm implemented is for checking whether a Conjunctive Query q has a Fractional Hypertree Decomposition $(\mathcal{T}, \mathcal{B}, \gamma)$ of width at most k + a small constant , algorithm that is described in this paper [20]. The algorithm is built upon the previous one, therefore they have a very similar structure, the only difference being the addition of one more set of guesses which corresponds to hyperedges that have a weight

smaller than 1. This chapter will have a similar structure to the previous one, where I will present the intuitive description of the algorithm and then the details of my implementation, along with the alterations that have been done.

Before presenting the pseudocode of the algorithm and the details of my implementation, let us define a few concepts taken from [20] which are essential for the way the algorithm works.

Definition A Fractional Hypertree Decomposition $(\mathcal{T}, \mathcal{B}, \gamma)$ of a hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{E})$ is in fractional normal form if and only if it respects the following requirements for any node r $\in \mathcal{V}(\mathcal{T})$ and each node s that is child of r:

- there is only one [$\mathcal{B}_r$]-component, C_r, such that all the vertices that appear in the bags of the nodes belonging to the subtree rooted in s, $\mathcal{T}_s$ are included in the set defined by the reunion the of the vertices in C_r and the intersection of the bags belonging to r and s, $\mathcal{B}(\mathcal{T}_s) = C_r \cup \mathcal{B}_s \cap \mathcal{B}_r$.
- there are no common vertices between the bag of s and the component from the previous requirement, $\mathcal{B}_s \cap [C_r] = \emptyset$.
- the intersection between the set of vertices defined by the guards of s and the vertices in the bag of r is included in the set of vertices defined by the bag of s, $\mathcal{B}_{\gamma_s} \cap \mathcal{B}_r \subseteq \mathcal{B}_s$.

Which is just a direct adaptation of the concept of a hypertree decomposition in normal form. At the same time, the authors define another concept which binds the run-time of the algorithm to be polynomial in the worst case.

Another useful concept presented in [20] which allows this algorithm to have a polinomially bounded run-time is that of a Fractional Hypertree Decomposition that has a c-bounded fractional part. As already mentioned, a fractional hypertree decomposition uses a labelling function $\gamma : e \in \mathcal{E}(\mathcal{H}) \rightarrow [0,1]$ which assigns to each edge in the hypergraph a fractional weight between 0 and 1. Let us now define the sets $C_1(s) = \{e \in \mathcal{E}(\mathcal{H}) | \gamma_s(e) = 1\}$ and $C_2(s) = \{e \in \mathcal{E}(\mathcal{H}) | \gamma_s(e) < 1\}$. Intuitively, this means we split the hypergedges which are part of the set of guards of a node s into two subsets, the hyperedges that have a weight of 1 and the hyperedges that have a weight strictly less than 1. Let a Fractional Hypertree Decomposition have a c-bounded fractional part if and only if for any node s in $\mathcal{V}(\mathcal{T})$ the number of vertices covered by hyperedges which have a weight less than 1 is at most c, $|\{var(e) | e \in C_2(s)$, where var(e) denotes the vertices covered by $\gamma\}| \leq c$.

Let $\mathcal{F} = (\mathcal{T}, \mathcal{B}, \gamma)$ be a FHD of a Conjunctive Query q. Taking the definition directly from [20], we say that $\mathcal{F}$ respects the weak special condition if for every node $s \in \mathcal{V}(\mathcal{T})$ the following condition holds: $\mathcal{B}(C_1) \cap \mathcal{V}(\mathcal{T}_u) \subseteq \mathcal{B}_u$. This is the same condition that allows the computation of Hypertree Decomposition to be done in polynomial time, but extended to the Fractional Hypertree Decomposition. Intuitively, it means that we only require the hyperedges which have a weight of 1 on the set of guards of s to respect the special condition, while the set C_2 of hyperedges with weight smaller

than 1 have no restriction.

Moreover, as the algorithm for computing a Fractional Hypertree Decomposition is heavily based on the algorithm for computing a Hypertree Decomposition presented in [5], it is as well described as for an alternating Turing machine, and parts such as guessing the set of guards and finding the [A]-Components based on the guards of a node will be computed in the same way as described in the implementation details for the Hypertree Decomposition algorithm.

Algorithm 3: (k, ϵ, c)-frac-decomp

input : Hypergraph H.
output: "Accept", if H has an FHD of width $\leq k + \epsilon$ with c-bounded fractional part and weak special condition
"Reject", otherwise.

Function f-decomp $(C_r, W_r$: *Vertex-Set*, R: *Edge-Set*$)$

- **begin** /* (1) Guess */
  - Guess a set $S \subseteq E(H)$ with $|S| = \ell$, s.t. $\ell \leq k + \epsilon$; /* (1.a) */
  - Guess a set $W_s \subseteq (V(R) \cup W_r \cup C_r)$ with $|W_s| \leq c$; /* (1.b) */
- **begin** /* (2) Check */
  - Check if $\exists \gamma$ with $W_s \subseteq B(\gamma)$ and $weight(\gamma) \leq k + \epsilon - \ell$; /* (2.a) */
  - Check if $\forall e \in edges(C_r)$: $e \cap (V(R) \cup W_r) \subseteq (V(S) \cup W_s)$; /* (2.b) */
  - Check if $(V(S) \cup W_s) \cap C_r \neq \emptyset$; /* (2.c) */
- **if** *one of these checks fails* **then Halt** and **Reject**; /* (3) */
- **else**
  - Let $\mathcal{C} := \{C \subseteq V(H) \mid C$ is a $[V(S) \cup W_s]$-component and $C \subseteq C_r\}$;
- **foreach** $C \in \mathcal{C}$ **do** /* (4) */
  - **if** f-decomp (C, W_s, S) *returns* **Reject then**
    - **Halt** and **Reject**
- **return Accept**;

begin /* Main */

- **return** f-decomp $(V(H), \emptyset, \emptyset)$

Figure 4.8: Fractional Hypertree Decomposition Pseudocode presented in [20] at page 44

Let us break down the algorithm presented in figure 4.8 and present intuitive description of the parts.

1. First step: As is the case with the Hypertree Decomposition algorithm presented in the previous section, the first step deals with the 'guess' of two sets. S is the set of hyperedges which have a weight of 1 with respect to the labelling function of the current node γ_s and W is the set of vertices which are covered by hyperedges which have a weight smaller than 1. Because the algorithm finds a Fractional Hypertree Decomposition that has a c-bounded fractional part, the size of W is bounded by the input parameter c.
2. Second step: The second step consists of mostly the same checks as the previous

algorithm and is even affected by the same mistake where they allow the intersection $e \cap (V(R) \cup W_r)$ to yield the null set as a result, this mistake is described in more details in the previous section and will not be repeated here. However, the FHD algorithm contains one more vital check which is non-trivial and presented great difficulty in transposing it from a pure theoretical environment to a real implementation. Let us name this step as 2a), the same way it is defined in the paper [20]. The check at the 2a) step ensures that there exists a fractional cover γ_s such that the weight of γ_s is at most k-l+ε, where k is the FHW of the hypergraph and l is the cardinality of S, and that every vertex in W is covered by γ, this means that for every vertex in W, the sum of weights of the hyperedges covering it is at least 1.

Intuitively, as S represents the set of hyperedges of weight 1, it means that the weight of S is already l, where l is the cardinality of S. Therefore, if we want to include more vertices by appointing weights smaller than 1 to hyperedges, we need to make sure that there exists such a fractional cover and that the selected fractional cover will not increase the total width of γ_s.

3. Third step: This step deals with finding the next [A]-Components that will be explored for discovering the decompositions of the next nodes. The only difference from the previous algorithm being that instead of having just var(S) as the set holding the vertices covered by the guards of the current node, we have $var(S) \cup W_s$. That is, the set of the vertices covered by the hyperedges with weight 1 reunited with the set of vertices covered by hyperedges with fractional weights. We can now be sure that there exists such a fractional cover for the vertices of W because of the check at the step 2a).

4. Fourth step: The last step is exploring all the possible components and check if they all respect the necessary conditions. As this belongs to the universal state of the alternating Turing machine, if any of the components fail the checks, the whole branch of the computation tree will fail.

One can view the algorithm as an improved version of the algorithm for Hypertree Decomposition presented in [5], where besides guessing the sets of the guards for the nodes of the decomposition, the algorithm also guesses sets of vertices covered by hyperedges with weight less than 1 in order to relax the total width of the decomposition. Another fact that should be noted is that by setting the parameter c which denotes the number of vertices covered by hyperedges with fractional weight to 0, the algorithms are semantically equivalent.

4.4.1 Personal Contributions to the FHD algorithm

In a similar manner to the previous section, I will present here the details of my implementation of the FHD algorithm along with accentuating the alterations from the canonical algorithm that were needed for the algorithm to work.

As the algorithm is described in the same high-level manner and is thought to be run on an alternating Turing machine, the implementation makes use of the same programming paradigm of Mutual Recursion [18] to simulate both the universal and the

existential components of the machine and has the same structure as the previous algorithm.

Moreover, in the canonical algorithm, the assignments of the sets S and W are based on random guesses, where S is sampled from the set of all hyperedges of the hypergraph and W is sampled from the vertices in the current component or which have been used in the guards of the parent. My implementation makes use of the same optimization as in the previous algorithm, where I draw the samples for S only from the hyperedges that cover the vertices of the current $[C_r] - Component$. This minimizes the total number of possibilities and eliminates the unnecessary work done by the algorithm.

Algorithm 2: Fractional Hypertree Decomposition(Existential part)

Input: a hypergraph G, a natural number k representing the hypertree width, an [A]-Component object $Comp_r$, a set $Guards_r$ representing the guards of the parent node, a set $Vertices_r$ representing the vertices covered by the guards and the set S of guessed guards, c which is the bound for W, W_r which are the vertices covered by hyperedges with fractional weights of the parent node

. **Output:** a boolean that specifies if a hypertree decomposition of width at most k has been found and the decomposition

.

```
for S in all possible subsets of at most k hyperedges from the component Comp_r do
    for W in all possible subsets of at most c vertices from W_r ∪ Comp_r ∪ Vertices_r
     do
        if Fractional Hypertree Decomposition (Universal part)(G, C_i, S, W_s), c )
          == true then
            return true;
        end
    end
end
```

Figure 4.9: The existential part of my implementation of the Frational Hypertree Decomposition Algorithm

Because this algorithm has the same backbone as the previous one, it suffers from the same ambiguities as that one, which are related to the alternating nature of the algorithms, implemented using Mutual Recursion, to the bug in the check at step 2b) which allows the intersection to produce null sets, to the way in which the sets S and W are picked for each node using the same 'GetSets' class implemented by me, to the same unnecessary work done by considering the whole set of hyperedges as domain for sampling S from, instead of only using the hyperedges of the component and to the same lack of an official algorithm for detecting the [S]-Components that need to be explored for the next nodes which I have fixed by creating and implementing a modified version of the Depth-First Search algorithm presented in figure 4.7.

The only new ambiguity of this algorithm is the check at step 2a) in which the algorithm needs to check if there exists a fractional edge cover of width k-l, where l is the cardinality of S, which covers the vertices of W.

My implementation makes use of the 'getSets' class to compute the samples for W in an efficient manner as the class works similar to a generator [17], which means that the

Algorithm 1: Fractional Hypertree Decomposition(Universal part)

Input: a hypergraph G, a natural number k representing the hypertree width, an [A]-Component object $Comp_r$, a set $Guards_r$ representing the guards of the parent node, a set $Vertices_r$ representing the vertices covered by the guards and the set S of guessed guards, c which is the bound for W, W_r which are the vertices covered by hyperedges with fractional weights of the parent node

. **Output:** a boolean that specifies if a hypertree decomposition of width at most k has been found and the decomposition

.

1. Check if the following requirements hold:
   There is γ such that $W_s \subseteq \mathcal{B}(\gamma)$ and weight(γ) $\leq k - l + \varepsilon$
   $\forall C \in Comp_r$, var(C) $\cap (Vertices_r \cup W_r) \subseteq (var(S) \cup W_s)$
   $(var(S) \cup W_s) \cap C \neq \varnothing$
2. Get all the possible components for the children of this node
   listOfComponents $\leftarrow$ getPossibleComponents($Comp_r, (var(S) \cup W_s)$)
3. Compute the bag of vertices for this node according to the following relation:
   $\mathcal{B}(currentNode) = \gamma_s$ if this is the root of the tree or
   $\mathcal{B}(currentNode) = \mathcal{B}(\gamma_s) \cap \mathcal{B}_r \cup Comp_r$
4. Iterate through all the possible components and check if they all lead to a valid decomposition
   for C_i *in listOfComponents* **do**
   if *Fractional Hypertree Decomposition (Existential part)*(G, C_i, S, W_s)) == *false* **then**
   return false;
   end
   end
   return true ;

Figure 4.10: The universal part of my implementation of the Fractional Hypertree Decomposition Algorithm

class will not return a list with all the possible subsets, but will keep the current state of computation internally and return the next state at a function call. This dramatically reduces the memory used by the algorithm and binds it polynomially with respect to the input.
After obtaining the sample for W, the algorithm needs to check the step 2a), which verifies that there exists a fractional edge cover that can cover the vertices of W. Instead of computing all the possible fractional covers and check whether they cover the vertices in W, the code spans a sub-hypergraph that contains only the vertices of W and the hyperedges that cover them, as the problems are equivalent but finding an edge cover for the sub-hypergraph eliminates a lot of redundant work.

After spanning a sub-hypergraph based on W, I use a linear equation solver [8] to define the requirements of the fractional edge cover: each hyperedge needs to have a weight of at least 0 and every vertex in the spanned sub-hypergraph needs to be covered by a sum of weights of at least 1, and the target of the solver: to minimize the total weight of the fractional cover. While computing the sub-hypergraph, the algorithm keeps track of the original indices of the hyperedges so the correct bags of vertices and guards can be formed and returned if a decomposition is found. The complexity of the linear solver is $O(N^2)$ where N is the size of the input, so it is therefore polynomial and does not add an overhead to the time complexity of the original algorithm. If the solver finds any fractional cover that respects the requirements, a mapping of hyperedges indices to the weights assigned to them is returned and the 2a) step in the algorithm is passed.

Let us examine the example in figure 4.11 to display how this part works on an intuitive level.

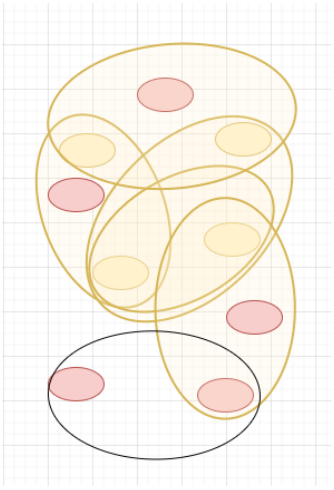

Figure 4.11: Example of sub-hypergraph

Let us assume that all the yellow vertices are within the set W and therefore, the algorithm needs to use them to span a sub-hypergraph. The algorithm will select all the hyperedges that cover at least one vertex from W, which are represented in the example with the yellow hue and after computing the sub-hypergraph, it will use it to create the requirements for the solver and discover any fractional hyperedge cover that respects the requirements specified in step 2a).

Chapter 5

Testing and Evaluation

The pieces of work on which my implementations are based, namely [5] and [20], deal only with the purely theoretical aspect of the decompositions, with few details about implementations and no explicit directions for testing the actual implementation. Thus, the testing that has been done for this project is based on examples of hypergraphs and their decompositions from works such as [5], [20], [9] as well as randomly sampled tests from [19] which is a benchmark in the domain of Hypertree Decompositions.

Because there can exist mutliple Hypertree decompositions of minimal width k for the same Conjunctive Query and because an automatic method of checking whether a Hypertree Decomposition is correct would use the same checks as the actual algorithm for generating the result and thus, it would perpetuate the same potential bug in the code, the testing for checking the actual decompositions has been done manually in order to guarantee the correctness of the results.

All the auxiliary methods and classes have been tested using JUnit tests which cover semantic edge-cases that have been collected through a thorough study of the existential material on the topic of Hypertree Decompositions. Apart from ensuring that the test suite covers a solid base of edge-cases, I have also picked code coverage as the metric to ensure that my implementations have been thoroughly tested.

The unit tests have also served as a regression testing suite which ensured that every new bug introduced with a change in the structure of the code will be detected. This was hugely beneficial as the algorithms have suffered many transformations throughout the course of several months of development.

5.1 Testing auxiliary classes

Let us name every class and method that is involved directly in the flow of the algorithm, apart from the methods that have been described in the pieces of work [5] and [20] as an auxiliary class, respectively, auxiliary method.

As the implementation of the main methods required the implementation of an extensive code base of auxiliary classes and methods, I deemed that end-to-end testing is not sufficient to catch all the possible bugs that can appear in a complex system. Therefore, I have implemented unit tests using JUnit4 that provide a solid code coverage in order to ensure the excellent quality of the implementation. The complete test suite can be analysed in the appendix A of the report or in the attached code files.

The test suite ensures that all auxiliary classes and methods behave the way they are intended to, even against malicious inputs, such as:

- Creating a hypergraph with 0 nodes and 0 vertices and trying to span a sub-hypergraph from it, as well as spanning an empty, partial or a sub-hypergraph equivalent with the original one.
- Initializing the generator class 'GetSets' with a negative parameter k which defines the maximal length of the subsets.
- Running the method that returns the available components given a set of guards on an empty hypergraph or with an empty set of guards.
- Covering the cases when the linear solver can and cannot cover all the vertices of the spanned sub-hypergraph.

These unit tests aided me in discovering subtle bugs in the initial pseudo-codes such as the bug discovered in the step 2a) of the algorithm described here 4.3 in which the intersection of the component with the vertices covered by the guards of the parent of the current node cannot yield the null set.

The development and testing of the project have been done using Intellij IDE and JUnit-Jupiter, which provides tools for class, method and code coverage of the testing suite.

Class	class coverage	method coverage	line coverage
Component	100%	100%	100%
Hypergraph	100%	100%	100%
GetSets	100%	100%	98%
SolverFED	100%	100%	100%
HDSolver	100%	57%	36%
FHDSolver	100%	55%	31%

The scores for the classes HDSolver and FHDSolver are lower because they contain the main algorithms which are not suited for automatic testing for the reasons listed above. Therefore, only the auxiliary methods in those classes have been tested using JUnit tests.

5.2 Evaluation Main Algorithms

I will now present the results of the Hypertree Decomposition and Fractional Hypertree Decomposition algorithms on several examples that I have handpicked from various re-

search papers and which I believe are complex enough to cover all the edge cases of the algorithm. Moreover, I will also present the results of a few samples picked at random uniformly from Hyperbench [19] and present the time the algorithms take to provide an answer. Unfortunately, a comparison with other algorithms in Hyperbench cannot be done, as the platform only accounts whether the algorithm took more than an hour to finish running or not.

This section shows the results of end-to-end testing, therefore, every class and method presented so far is being used in the complex flow of the algorithms, from the creation of the hypergraph to the printing of the results. The results have been analysed manually by me for the same reasons as explained at the beginning of this chapter and the raw output of the algorithm on a couple of examples will also be presented in this section.

Let us start with an easy input and work our way up to more complex examples. Thus, the first input is a hypergraph from the Loomis Whitney family, with k = 3, which has also been presented in figure 2.3.

```
true
Bags
1 2
1 2 3
Guards
0
1 2
The execution took PT0.0129984S seconds
Process finished with exit code 0
```

Figure 5.1: Raw output of the Hypertree Decomposition algorithm on the Loomis Whitney k=3 hypergraph

```
true
The vertices in the bags are 1 2 3
The guards are 0 has weight 0.5, 1 has weight 0.5, 2 has weight 0.5,
The execution took PT0.0560003S seconds

Process finished with exit code 0
```

Figure 5.2: Raw output of the Fractional Hypertree Decomposition algorithm on the Loomis Whitney k=3 hypergraph

The first thing to notice is the widths of the decompositons. In the case of the Hypertree Decomposition, this can be measured by the maximal number of hyperedges in the set of guards of a node. For our input, this is 2.

In the case of Fractional Hypertree Decomposition, the width of the decomposition is given by the sum of the weights on each hyperedge in the set of the guards of a node and for out input, this is 1.5.
As expected, the FHW is smaller than HW for the input Conjunctive Query but what is surprising is that the time difference is not significant for this size of input.

Let us take as input the hypergraph which has been presented in the figure 4.4 and which was exploiting the bug of the null intersection in the step 2a) of the Hypertree Decomposition algorithm and step 2b) of the Fractional Hypertree Decomposition algorithm.

One can see that by not allowing the intersection at step 2a), respectively 2b) to be null, the algorithms correctly identify the graph as being cylclic (due to the fact that

```
true
Bags
1 2
1 2 3
3 4
Guards
0
1 2
3
The execution took PT0.0139997S seconds
Process finished with exit code 0
```

Figure 5.3: Raw output of the Hypertree Decomposition algorithm on the altered Loomis Whitney hypergraph for k=3

```
true
The vertices in the bags are 1 2 3
The vertices in the bags are 3 4
The guards are 0 has weight 0.5, 1 has weight 0.5, 2 has weight 0.5,
The guards are 3 has weight 1.0,
The execution took PT0.0790109S seconds

Process finished with exit code 0
```

Figure 5.4: Raw output of the Fractional Hypertree Decomposition algorithm on the altered Loomis Whitney hypergraph for k=3

the width is greater than 1) and computes a decomposition of width 2, respectively 1.5 for it.

All the inputs so far have been rather small, with just a few vertices and hyperedges, but if we are to go the other extreme and test the algorithm on a Loomis Whitney hypergraph with k=10, we will notice significant differences in both the run time of the algorithms and the final widths of the decompositions.

```
true
Bags
1 2 3 4 5 6 7 8 9
0 1 2 3 4 5 6 7 8 9
Guards
0
1 2
The execution took PT0.0130625S seconds
Process finished with exit code 0
```

Figure 5.5: Raw output of the Hypertree Decomposition algorithm for Loomis Whitney hypergraph for k = 10

```
true
The vertices in the bags are 0 1 2 3 4 5 6 7 8
The vertices in the bags are 0 1 2 3 4 5 6 7 8 9
The guards are 0 has weight 0.125, 1 has weight 0.125, 2 has weight 0.125,
The guards are 0 has weight 0.11111111111111105, 1 has weight 0.1111111111
The execution took PT2.9055731S seconds

Process finished with exit code 0
```

Figure 5.6: Raw output of the Fractional Hypertree Decomposition algorithm for Loomis Whitney hypergraph for k = 10

Even though the decompositions consist of the same hyperedges and the same nodes, the widths start to have drastic differences, from a widht of 2 for the Hypertree Decomposition to a width of approximately 0,11 for the Fractional Hypertree Decomposition. At the same time, the FHD algorithm takes almost 300 times more time than HD for Loomis Whitney hypergraph for k=10, if we pick k=100, the HD algorithm takes 0,1 seconds to find a solution, while the FHD algorithm cannot find one in more than an hour.

One of the most complex hypergraphs that have been provided as input is the hypergraph presented at page 20, figure 4 in [20].

The example in figure 5.7 is sufficiently complex to be non-trivial even for humans, especially for finding a minimal Fractional Hypertree Decomposition and has been part of the regression testing over the development of the algorithms.

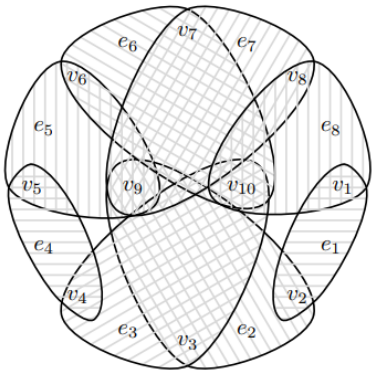

Figure 5.7: Hypergraph presented in [20]

```
true
Bags
1 2
1 2 3 8 9 10
3 4 7 8 9 10
4 5 6 7 9 10
Guards
0
1 7
2 6
3 4 5
The execution took PT0.0130011S seconds
Process finished with exit code 0
```

Figure 5.8: Raw output of the Hypertree Decomposition algorithm

For this example, the FHD algorithm was executed with parameters k = 2.5 and c = 3. It is worth mentioning that the algorithms return the earliest found correct decomposition, even if that means not computing the whole recursive tree, but they will exhaust all the possibilities for a lower value of k before increasing the width of a possible decomposition. Even for this example, the FHD algorithm with parameters k=2 and c=10 ran for an hour without finding a correct decomposition.

```
true
The vertices in the bags are 6 7 9
The vertices in the bags are 3 4 6 7 9 10
The vertices in the bags are 2 3 7 8 9 10
The vertices in the bags are 1 2 7 8 10
The vertices in the bags are 4 5 6 7 9
The guards are 4 has weight 0.5, 5 has weight 0.5, 6 has weight 0.5,
The guards are 2 has weight 1.0, 4 has weight 0.5, 5 has weight 0.5, 6 has weight 0.5,
The guards are 1 has weight 1.0, 5 has weight 0.5, 6 has weight 0.5, 7 has weight 0.5,
The guards are 0 has weight 1.0, 5 has weight 0.5, 6 has weight 0.5, 7 has weight 0.5,
The guards are 3 has weight 1.0, 4 has weight 0.5, 5 has weight 0.5, 6 has weight 0.5,
The execution took PT8.593S seconds

Process finished with exit code 0
```

Figure 5.9: Raw output of the Fractional Hypertree Decomposition algorithm with c = 3

Chapter 6

Conclusion and Future Work

The work presented in this paper has been based almost exclusively on the pseudo-codes and theoretical background presented in General and Fractional Hypertree Decompositions: Hard and Easy Cases [20] and Hypertree Decompositions and Tractable Queries [5]. With regards to the goals of the project specified at the beginning, the following have been successfully accomplished:

- Analysed existing literature and transposed it into an easy-to-read form.
- Implemented an algorithm for finding whether a Conjunctive Query q allows a Hypertree Decomposition of width at most k in asymptotic polynomial time.
- Pioneered one of the first implementations for a Fractional Hypertree Decomposition algorithm that checks whether a Conjunctive Query q allows a c-bounded fractional part decomposition of width bounded by k + a small constant ε.
- Tested both algorithms on relevant examples from the existing literature, as well as randomly sampled examples from Hyperbench.

Moreover, I succeeded in critically assessing the canonical pseudo-codes and correct two bugs that were disrupting the logical flow, to come up with an original solution for a sub-task that is vital for both algorithms and to optimize the practical run-time of the algorithms through a non-trivial observation.

Although the algorithms have a large constant factor regarding the theoretical asymptotic complexity, the Hypertree Decomposition algorithm proved to be sufficiently fast on every input to be useful in practice. In spite of the significantly larger run-time overhead, the Fractional Hypertree Decomposition algorithm proved to be more powerful in generalising query acyclicity, but it is less trivial to use because of the multiple needed parameters and for certain inputs, the difference between the widths of the decompositions were not sufficient.

In terms of future work which was either outside of the direct scope of this work or exceeded the resources available at this time, there are a few notions worth mentioning:

- The structures of both algorithms can be viewed as a Depth-First Search into

the computational tree of the problem. It could be worth experimenting with modifying this structure such that it will be based on Breadth-First Search so it can benefit from the property that the first path found to a solution is guaranteed to be the shortest one. Obviously, an implementation based on BFS will need to be careful at memory management and will still need to keep the alternating Turing Machine behaviour.

- The selection of the parameter c for the Fractional Hypertree Decomposition algorithm is non-trivial and it has a massive impact on both the time performance of the algorithm and on the results. A method of reducing the field of possible values for c could prove to be helpful.
- As for the current implementation, the Fractional Hypertree Decomposition algorithm uses a linear solver program to compute a fractional cover for a sub-hypergraph spanned by a set of vertices. This makes the algorithm significantly slower than the Hypertree Decomposition algorithm. Therefore, it could be worth researching into faster ways of checking that the needed requirements are satisfied.
- In this project, I have implemented and tested the behaviour of two algorithms for decomposing a Conjunctive Query. A research into whether the overhead of applying these algorithms to a Conjunctive Query is worth compared to the time needed to run the original query will concretise the practical utility of query decomposition.

Bibliography

[1] Anand Rajaraman Chandra Chekuri. Conjunctive query containment revisited.

[2] Maryam Hazman Mahmoud Abd ElLatif Fayed F. M. Ghaleb, Azza A. Taha and Mona Abbass. On quasi cycles in hypergraph databases.

[3] Christopher Ré Semih Salihoglu Jeffrey D. Ullman Foto Afrati, Manas Joglekar. Gym: A multiround join algorithm in mapreduce.

[4] Gianluigi Greco Georg Gottlob. Decomposing combinatorial auctions and set packing problems.

[5] Nicola Leone Georg Gottlob and Francesco Scarcello 2002. Hypertree decompositions and tractable queries. j. comput. syst. sci. 64, 3 (2002), 579–627.

[6] Nysret Musliu Marko Samer Francesco Scarcello Georg Gottlob, Martin Grohe. Hypertree decompositions: structure, algorithms, and applications.

[7] Thomas Schwentick Georg Gottlob, Zoltán Miklós. Generalized hypertree decompositions: Np-hardness and tractable variants.

[8] Michael Schober Hannes Planatscher. Scpsolver - an easy to use java linear programming interface.

[9] Martin Grohe Isolde Adler, Georg Gottlob. Hypertree width and related hypergraph invariants.

[10] P.D Seymour Neil Robertson. Graph minors. ii. algorithmic aspects of tree-width.

[11] P.DSeymour NeilRobertson. Graph minors. x. obstructions to tree-decomposition.

[12] Carl Kingsford Rob Patro. Predicting protein interactions via parsimonious network history inference.

[13] Paul Seymour Sang-il Oum. Approximating clique-width and branch-width.

[14] Seagate. Data age 2025.

[15] C. L. Philip Chen Shulong Tan. Using rich social media information for music recommendation via hypergraph model.

[16] Wikipedia. Constraint satisfaction problem.

[17] Wikipedia. Generator.

[18] Wikipedia. Mutual recursion.

[19] Davide M Longo Wolfgang Fischl, Georg Gottlob and Reinhard Pichler. Hyperbench: A benchmark and tool for hypergraphs and empirical findings.

[20] REINHARD PICHLER WOLFGANG FISCHL, GEORG GOTTLOB. General and fractional hypertree decompositions: Hard and easy cases.

[21] M. YANNAKAKIS. Algorithms for acyclic database schemes – vldb 1981.