Wesleyan ♦ University

An Exploration of Nest-Set Width and its Applications

By
Linnea Caraballo
Faculty Advisor: Dr. Karen L. Collins

A Thesis submitted to the Faculty of Wesleyan University in partial fulfillment of the requirements for the degree of Master of Arts

Middletown, Connecticut May 2024

Abstract

Hypergraph acyclicity has been extended to several notions of acyclicity in hypergraphs: alpha acyclicity, beta acyclicity, and gamma acyclicity. It has been shown that many fundamental database problems that are NP-hard are tractable when the underlying hypergraph is acyclic. Despite the usefulness of all degrees of acyclicity, beta acyclicity, and gamma acyclicity have been given far less attention, particularly when it comes to generalizing these tractable results beyond acyclicity. We will explore a generalization of beta acyclicity introduced by Lanzinger called nest-set width and expand upon this discussion by providing observations gleaned from studying a large data set of hypergraphs.

Acknowledgment

I would like to give thanks and appreciation to all those who have given me support over my educational journey. First and foremost, I would like to thank my parents for their unwaverying support and encouragement. I would also like to extend my appreciation to Professor Karen Collins, for her mentorship and guidance as I explored the world of graph theory. I would also like to thank my undergraduate professors at Sacred Heart University for sparking my passion and love of mathematics and encouraging me through the graduate application process and my time as a graduate student at Wesleyan University. Lastly, I would like to thank my friends and fellow graduate students for their their unwavering support, guidance, and camaraderie. Without everyone's support, this thesis would not have been possible.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Graph theory is the field of study that deals with graphs. Graphs are mathematical structures used to model pairwise relationships between objects. Graphs are widely used in biology, chemistry, physics, and computer science. However, there are times when we want to model relationships between more than just two objects. Hypergraphs are a generalization of graphs that model the relationships between sets of objects. Expanding the theorems and concepts found in graph theory to hypergraphs forms a new area of study called hypergraph theory.

One of the most interesting properties of hypergraphs is acyclicity. The notion of hypergraph acyclicity has been studied extensively and extended to various degrees. In order of decreasing generality, we have alpha acyclicity, beta acyclicity, and gamma acyclicity, each of which admits various characterizations. The most explored is alpha acyclicity which has been shown to

have useful applications in database theory as described by Beeri et al. [2].

Acyclicity of hypergraphs is relevant in the study of complexity reasoning. Many NP-hard problems become tractable when restricted to acyclic instances. For example, the evaluation of conjunctive queries is a problem in database theory that is known to be NP-hard in general. However, it becomes tractable when the underlying hypergraph structure of a query is alpha acyclic as demonstrated by Yannakakis [31]. Further restricting our query to be beta acyclic allows for tractability of conjunctive queries with negation and SAT, which remain NP-hard for alpha acyclic hypergraph structures as described by Brault-Baron [3]. However, while generalizations of alpha acyclicity have been explored, generalizations of beta acyclicity have received little attention and many generalizations do not maintain the tractability of beta acyclic instances.

In recent work, Lanzinger [24] introduced a new generalization of beta acyclicity called nest-set width. Nest-set width builds off of nest-point elimination orderings of hypergraphs, a common strategy in determining beta acyclicity of hypergraphs, by extending nest-points to nest-sets. More importantly, it has been shown that when bounded by nest-set width boolean conjunctive queries with negation and SAT are tractable. We aim to explore the acyclicity of hypergraphs and nest-set width and their applications, building off of the work of Beeri et al. and Lanzinger.

This thesis is structured in the following way: in Chapter 2 we will provide background information on graph theory, algorithms, and databases. Chap-

ter 3 will expand on acyclicity of hypergraphs, Chapter 4 will define nest-set width and we will discuss new observations, and Chapter 5 explores the applications of acyclicity and nest-set width in database theory, ultimately leading to us showing tractability of conjunctive queries with negation and SAT.

Chapter 2

Background

2.1 Graph Theory

Throughout this section, we will use the notation and definitions from Diestel's book *Graph Theory* [12] unless otherwise specified. A *graph* is a mathematical structure $G = (V, E)$ where V is a set of vertices, or objects, and E is a set of edges, or relations, such that each edge is a 2-element subset of V. We will denote an edge as an unordered pair of vertices, $e = \{v_i, v_j\}$ where $v_i, v_j \in V$ and $e \in E$. If G is not obvious we denote the set of vertices as $V(G)$ and the set of edges as $E(G)$. For simplicity, we will work with simple graphs, graphs whose edges have no direction, no repeated edges, and no loops. A loop in a graph G is an edge $e = \{v_i, v_i\}$ where $v_i \in V(G)$. Additionally, we will not assign a direction to our edges.

The *order* of a graph, denoted $|G|$, is the number of vertices in G, and the

size of a hypergraph is the cardinality of E, denoted $|E|$. Each vertex has a notion of size called the *degree*, denoted $\deg_G(v)$ or $\deg(v)$ if G is obvious. The degree of a vertex is the number of edges at v, denoted $|E(v)|$. A vertex that has no edges, degree 0, is called an *isolated vertex*.

Since a graph is defined by a vertex and edge set we can find subsets of the vertex and edge set, leading us to the definition of a subgraph.

Definition 2.1. $G' = (V', E')$ is a *subgraph* of $G = (V, E)$ (and G is a *supergraph* of G'), if $V' \subseteq V$ and $E' \subseteq E$. Note that an edge in $E(G')$ cannot contain a vertex that is not in $V(G')$.

To continue our discussion on graphs it is important to develop the language to properly describe relations between vertices and edges.

Definition 2.2. A vertex v is *incident* with an edge e if $v \in e$. The two incident vertices of an edge are called the *endvertices* or *ends*, and an edge joins its endvertices.

Definition 2.3. Let G be a graph and $v_1, v_2 \in V$. We say v_1 and v_2 are *adjacent*, or *neighbors*, if $\{v_1, v_2\}$ is an edge in G. Observe that the number of neighbors of a vertex, v, is equal to $\deg(v)$. Two edges $e_1, e_2 \in E$ such that $e_1 \neq e_2$ are *adjacent* if they have an end vertex in common. We call pairwise non-adjacent vertices or edges *independent*.

Definition 2.4. A *path* is a non-empty graph $P = (V, E)$ where $V = \{v_0, v_1, v_2, \ldots, v_k\}$ and $E = \{\{v_0, v_1\}, \{v_1, v_2\}, \ldots, \{v_{k-1}, v_k\}\}$ such that each

v_i is distinct. We say v_0 and v_k are linked by P and are the endvertices, or ends, of P. The vertices $v_1, \ldots, v_{k-1}$ are the *inner vertices* of P. The number of edges in a path is its *length* and a path of length k is denoted $P^k = (v_0, v_1, v_2, \ldots, v_k)$. A graph G is *connected* if it is non-empty and any two of its vertices are linked by a path in G.

Definition 2.5. A path $P = (v_0, v_1, \ldots, v_k)$ with $k \geq 3$, is a *cycle* if $v_0 = v_k$. The *length* of a cycle is the number of distinct edges or vertices, and a cycle of length k is denoted C^k and called a k-cycle. A graph that contains no cycles is called *acyclic*.

Lemma 2.1 ([4])**.** *A graph in which each vertex has exactly two neighbors has a cycle.*

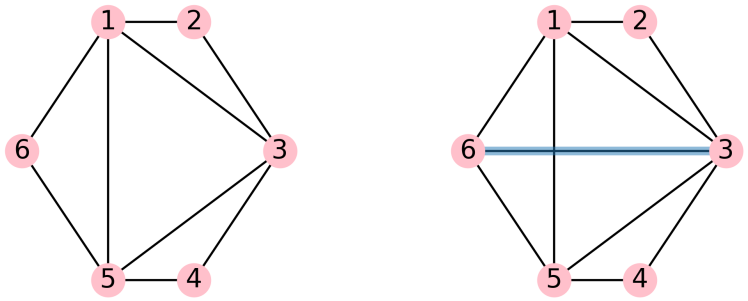
Proof. Let G be a graph where every vertex has exactly two neighbors and let $P = (v_1, \ldots, v_{k-1}, v_k)$ be the longest path of G. We know $\deg(v_k) = 2$, so there exists a $v_\ell \in V$ that is adjacent to v_k such that $v_\ell \neq v_{k-1}$. If $v_\ell \notin P$ then this contradicts the fact that P is the longest path, so v_ℓ must be in P. This means there exists a path $(v_\ell, \ldots, v_{k-1}, v_k, v_\ell)$ which is a cycle. □

Throughout this paper, we will refer to an important family of graphs called chordal graphs, and a subfamily of chordal graphs called strongly chordal graphs. Before we define a chordal and strongly chordal graph we must first define a chord and an odd chord.

Definition 2.6 ([19])**.** A *chord* is an edge that joins two non-consecutive, independent, vertices in a cycle. If $\{v_i, v_j\} \in E$ is a chord and one of i and j

is even and the other odd, we call $\{v_i, v_j\}$ an *odd chord*. In Figure 2.1a the edge $\{1, 5\}$ is a chord of the cycle $(1, 3, 5, 6, 1)$ and in Figure 2.1b the edge $\{3, 6\}$ is an odd chord of the cycle $(1, 2, 3, 4, 5, 6, 1)$.

Definition 2.7 ([19])**.** A *chordal graph* is a graph where every cycle of four or more vertices has a chord. A *strongly chordal graph* is a chordal graph where every even cycle of length 6 or more has an odd chord.



(a) Chordal graph, G'.

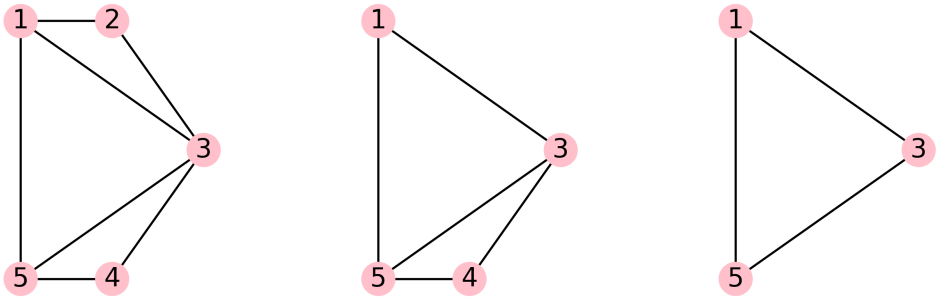
(b) Strongly chordal graph, G.

Figure 2.1: [30] In Figure 2.1a we can see that every cycle of 4 or more has a chord, however, none of these are odd chords since they connect vertices that are an even length away. In Figure 2.1b we can see the edge $\{3, 6\}$ is an odd chord in a cycle of length 6. Since there is only one cycle of length 6 and it has an odd chord, the graph G is strongly chordal.

We now understand what makes a graph chordal and strongly chordal, however the definitions as given result in a tedious identification process. A well-known characterization of chordal graphs observed by Fulkerson and Gross [16] gives us a systematic way to identify chordal graphs. A graph is chordal if and only if it has a perfect elimination ordering. Before we define a perfect elimination ordering we must first understand simplicial vertices.

Definition 2.8 ([19])**.** Let G be a graph. A vertex $v \in G$ is *simiplicial* if its neighborhood $N(v)$, the set of all neighbors of v, is a *clique* which means any two neighbors of v are connected by an edge in G. An ordering $(v_1, v_2, \ldots, v_n)$ of vertices of a graph G is a *perfect elimination ordering* (PEO) of G if each vertex v_i is simplicial in the induced subgraph $G_i = G[v_i, v_{i+1}, \ldots, v_n]$. Where the induced subgraph consists of all edges whose endvertices are in the set $\{v_i, v_{i+1}, \ldots, v_n\}$.

Example 2.1.



(a) $G' \setminus 6$. (b) $G' \setminus \{2\}$. (c) $G' \setminus \{2, 4\}$

Figure 2.2: Perfect elimination ordering of Figure 2.1a.

Let us look at the chordal graph in Figure 2.1a. Observe that vertex 6 is a simplicial vertex since $N(6) = \{1, 5\}$ which is an edge. So we can remove 6 and its incident edges, Figure 2.2a. Observe that 2 is a simplicial vertex of $G'_2 = G[2, 3, 4, 5, 1]$, so we remove 2 and its incident edges, Figure 2.2b. Then we can remove 4, Figure 2.2c, leaving us with a complete graph, so every vertex is simplicial and the order of removal does not matter. This leaves us with the following perfect elimination ordering: (6, 2, 4, 1, 3, 5).

Because strongly chordal graphs are chordal, every strongly chordal graph will have a perfect elimination ordering, however, a PEO does not specify a chordal graph from a strongly chordal graph. We need to establish a new elimination ordering for strongly chordal graphs.

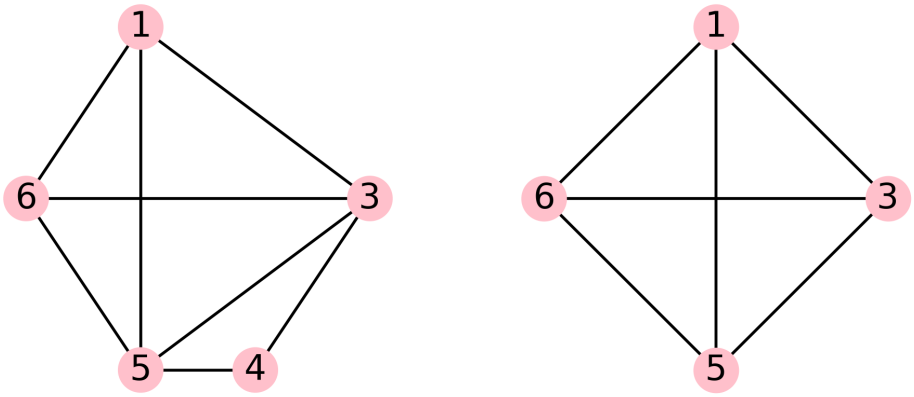
Definition 2.9 ([19])**.** Let G be a graph. A vertex $v \in V(G)$ is *simple* if the set of closed neighborhoods, $\{N[u] : u \in N[v]\}$ is linearly ordered with respect to set inclusion. A vertex ordering $(v_1, v_2, \ldots, v_n)$ is a *simple elimination ordering* if, for all $i \in \{1, 2, \ldots, n\}$, v_i is simple in $G_i = G[\{v_i, v_{i+1}, \ldots, v_n\}]$.

In Figure 2.1b we can see that 2 is a simple vertex since $\{N[1], N[2], N[3]\} = \{\{1, 2, 3, 6\}, \{1, 2, 3\}, \{1, 2, 3, 4, 5, 6\}\}$ is linearly ordered by set inclusion. Observe that in Figure 2.1b our vertex 2 is also a simplicial vertex, however in Figure 2.1a, 2 is not a simple vertex since $\{N[1], N[2], N[3]\} = \{1, 2, 3, 6\}, \{1, 2, 3\}\{1, 2, 3, 4, 5\}$ is not linearly ordered.

Example 2.2.

Let us look at our strongly chordal graph in Figure 2.1b. We know that 2 is simple, so we can remove it and its incident edges, Figure 2.3a. Observe that 4 is a simple vertex of $G_4 = G[\{4, 5, 6, 1, 3\}]$, so we can remove 4 and its incident edges, Figure 2.3b. Now we are left with a complete graph with 4 vertices so every vertex is simple and the order in which we eliminate them does not matter. Thus one simple elimination ordering is $(2, 4, 1, 3, 5, 6)$.

Chordal graphs are important because they are related to hypergraphs,



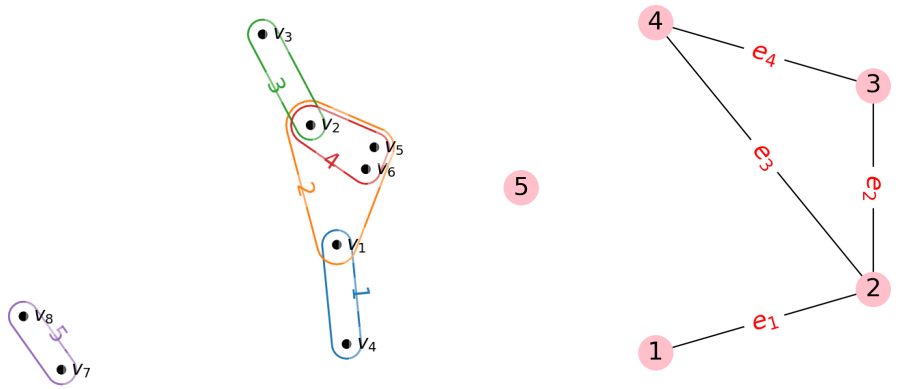
(a) $G \setminus 2$ (b) $G \setminus 4$

Figure 2.3: Simple elimination ordering of Figure 2.1b.

a generalization of graphs. A *hypergraph* H denoted $H = (V, E = (e_i)_{i \in I})$ on a finite set V is a family $(e_i)_{i \in I}$ of subsets of V called *hyperedges*, where I is a finite set of indexes. We denote V as $V(H)$ and E as $E(H)$ when H is not obvious. An example of a hypergraph is shown in Figure 2.4a. We call a hypergraph *simple* if $e_i \subseteq e_j$ implies $i = j$. For the rest of this section, we will be following definitions from Bretto's book *Hypergraph Theory: An Introduction* [6] unless otherwise stated.

Recall that a graph models the relationship between pair-wise objects. This allows us to model hypergraphs by modeling the relationship between hyperedges, edges in a hypergraph, and vertices in our hypergraph. The graphical representation of a hypergraph $H = (V, E = (e_i)_{i \in I})$ is the *line graph* or *intersection graph* denoted as $L(H) = (V', E')$ where $V' := I$ or $V' := E$ if H has no repeated hyperedges and $\{i, j\} \in E'$ $(i \neq j)$ if and only if $e_i \cap e_j \neq \emptyset$. An example of a hypergraph and its intersection graph can be

seen in Figure 2.4.



(a) Hypergraph. (b) Intersection graph of hypergraph.

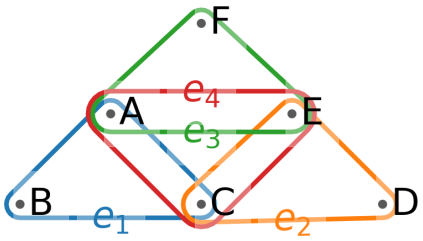
Figure 2.4: Each hyperedge in Figure 2.4a is represented by a vertex in Figure 2.4b. Our edges exist if the corresponding hyperedges overlap, i.e. share a vertex. So e_1 is an edge because v_1 is in our first and second hyperedge.

Because hypergraphs generalize graphs we can extend our definitions defined for graphs to hypergraphs. A hypergraph H' is a subhypergraph of a hypergraph H if $V' \subseteq V$ where

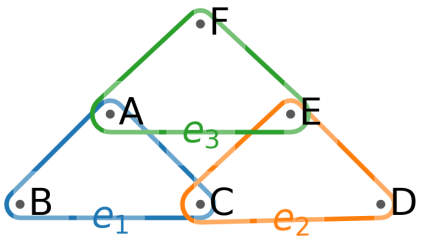
$$H' = \{V', E' = (e_j)_{j \in J}\} \text{ such that for all } e_j \in E', e_j \subseteq V'.$$

Note that we obtain H' by removing the hyperedges, and not the vertices. We cannot remove vertices independently from hyperedges since a vertex cannot be eliminated if a hyperedge that contains it still exists in H'.

For a set of vertices $V' \subseteq V$, $H(V') = (V', E')$ is the *induced subhyper-*



(a) Hypergraph, H.



(b) Subhypergraph of H.

Figure 2.5: [13] Note that our hypergraph in Figure 2.5b has all the hyperedges in Figure 2.5a except for e_4. Observe that all our vertices from Figure 2.5a remain in Figure 2.5b since they are all a part of other hyperedges. We could not remove C because C is in e_1 and e_2, so to remove C we would have to remove e_1 and e_2 along with vertices B and D.

graph where

$$E' = \{V(e_i) \cap V' \neq \emptyset : e_i \in E \text{ and either } e_i \text{ is a loop or } |V(e_i) \cap V'| \geq 2\}.$$

Two vertices in a hypergraph are *adjacent* if there is a hyperedge that contains both vertices. Two hyperedges are adjacent or *incident* if their intersection is not empty. In Figure 2.5a e_1 and e_2 are incident. A *path*, P, between two distinct vertices, x and y, in a hypergraph H is given by a vertex-hyperedge alternating sequence

$$P = (x = v_1, e_1, v_2, e_2, \ldots, v_s, e_s, v_{s+1} = y)$$

where $v_1, v_2, \ldots, v_{s+1}$ are distinct vertices, and $e_1, e_2, \ldots, e_s$ are distinct hyperedges, however it is possible that $v_1 = v_{s+1}$. And for all $i \in \{v_1, v_2, \ldots, v_{s+1}\}$, $v_i, v_{i+1} \in e_i$.

Now that we have the notion of a path, we can extend the notion of cycles and acyclicity to hypergraphs. However, because hyperedges can contain more than two vertices we have multiple non-equivalent definitions of cycles and acyclicity for hypergraphs. When looking at our original notion of a graph, $G = (V, E)$, all definitions of acyclicity for hypergraphs collapse to the acyclicity definition in Definition 2.5 through the intersection graphs. The three most common notions of acyclicity listed in decreasing order of generality are alpha acyclicity, beta acyclicity, and gamma acyclicity. We will explore acyclicity more in-depth in Chapter 3.

2.2 Algorithms

Throughout this section, we will follow the definitions and the notation from Gibbons book *Algorithmic Graph Theory* [17]. An *algorithm* is a finite sequence of rigorous instructions where there is no ambiguity in any instruction or the order in which the instructions are executed. Algorithmic efficiency relates to the amount of computational resources that the algorithm uses. In particular, we are interested in the *time-complexity* of algorithms, which we will refer to as just *complexity* as we are not concerned with space-complexity, the amount of memory space required. The *complexity* of an algorithm is

the number of computational steps it takes to change the input data into the desired output. We refer to the size of the input data as the *problem size*, and since we are working with graphs our problem size will be the size or order of a graph.

The complexity of an algorithm A is determined by a function that is dependent on the problem size, s, and denoted $C_A(s)$ or $C(s)$ if A is obvious. Because the complexity of an algorithm changes for each input, $C_A(s)$ only considers the *worst-case* complexity. So, $C_A(s)$ is the longest possible time that it could take an algorithm A to run.

The *order* of a function is the growth rate of the function. Given two functions F and G whose domain is the natural numbers we say F has a lower order than G if $F(n) \leq K \cdot G(n)$ where $K, n \in \mathbb{Z}^+$. We denote the order of F as $O(F)$ and the order of G as $O(G)$. F and G are of the same order if $F = O(G)$ and $G = O(F)$. We refer to $O(F)$ and $O(G)$ as big O notation.

Example 2.3. Let f be a function used to estimate the run time of an algorithm with input size n

$$f(n) = 7\log(n) + 15n^2 + 8n^3 + 4$$

Because we are only interested in when our n becomes large, we can ignore

low-order terms of our function. So we have

$$O(f(n)) = O(n^3)$$

Table 2.1: ([17]) These are different complexities ordered from smallest (most desirable, to largest (least desirable) where n is the size of our input.

Complexities where n is the Input Size			
Time-Complexities	Big O	$n = 2$	$n = 8$
Constant	$O(1)$	1	1
Logarithmic	$O(\log_2(n))$	2	3
Linear	$O(n)$	2	2^3
Linearithmic	$O(n \log_2(n))$	2	3×2^3
Quadratic	$O(n^2)$	2^2	2^6
Cubic	$O(n^3)$	2^3	2^9
Polynomial	$O(n^c)$, $c \in \mathbb{Z}$	2^c	2^{3c}
Exponential	$O(b^n)$, $b > 1$	b^2	b^8
Factorial	$O(n!)$	2	5×2^{13}

The complexity of algorithms helps us determine efficiency, however, it is important to note that the existence of an algorithm does not guarantee that the problem is solvable. There exist algorithms so inefficient that computation needs exceed the ability of modern-day computers. This is where it becomes important to differentiate between problems that can be solved in polynomial time and those that cannot. We call problems for which there exists a polynomial time algorithm P. A problem for which there is no known efficient algorithm, but that can be verified in polynomial time are called NP.

A decision problem is a computational problem that can be posed as a

yes-or-no question. Given two decision problems D_1, D_2 there is a *polynomial transformation* from D_1 to D_2 denoted $D_1 \propto D_2$, if the following hold:

(a) There exists a function $F(I)$ transforming any. instance I of D_1 to an instance of D_2 such that the answer to I with respect to D_1 is 'yes' if and only if the answer to $F(I)$ is 'yes' with respect to D_2.

(b) There exists an efficient algorithm to compute $F(I)$.

A problem D is *NP-complete*, if $D \in NP$ and if for every $D' \in NP$, $D' \propto D$.

We will now look at the first problem known to be NP-complete as proven by Cook [10]. The *satisfiability of conjunctive normal forms*, SAT, problem asks whether or not there exists a truth assignment that makes a boolean formula $C(v_1, v_2, \ldots, v_n)$ true. We will show the fixed-tractability of SAT based on a property called nest-set-width in Chapter 5.2.

2.3 Databases

To discuss databases we will follow Levene and Loizou's book *A Guided Tour of Relational Databases and Beyond* [25]. A *database* is an organized collection of logically interconnected data items. A *database management system* (DBMS) is a computer system responsible for the efficient storage and retrieval of data. A DBMS supports a data model, where a *data model* is made up of three parts: the *structure*, a collection of data structures, the *integrity*, a collection of general integrity constraints, and the *manipulative*,

a collection of rules.

Throughout this paper we will focus on the *relational data model*, dedicating a subsection to each of the three parts of our relational data model. First, we will discuss the data structure of our relational model which consists of a data schema, a collection of all the relation schemas in a database, and the database itself. Then we will explore the manipulative part where we will discuss the relational algebra before talking about the integrity part of our relational data model which consists of primary and foreign keys.

2.3.1 Data Structure

The relational model has one data structure, the relation. A *relation* is a set of tuples where each tuple represents some object that is distinguishable, more commonly referred to as an entity. Each table below, Table 2.2, 2.3, 2.4, 2.5, are examples of relations. Each column header is called an *attribute* and we denote the universe of attributes, $\mathcal{U}$, where $\mathcal{U}$ is a countably infinite set of attributes. The rows of our tables are *tuples*, where each tuple is part of a countably infinite set of constant values, called the underlying database domain, denoted $\mathcal{D}$. Given an attribute A in $\mathcal{U}$ the domain of A, DOM(A), is a subset of $\mathcal{D}$ which we will refer to as constants.

Example 2.4. The attribute SNAME represents the name of students modeled by the relation in Table 2.2, so SNAME $\in \mathcal{U}$. 'John' is a constant in SNAME and part of the tuple [500, 'John', 20, 'Malet St', 'CS', 'BSC', 'first']

Table 2.2: Relation over STUDENTS.

ID	SNAME	AGE	ADDRESS	DEPT	DEGREE	YEAR
500	John	20	Malet St	CS	BSC	first
623	Jane	22	Harold Rd	MA	BSC	fourth
522	Emma	18	Oxford St	MA	BSC	first
545	Tom	20	Queens Ave	BIOL	BSC	second

Table 2.3: Relation over DEPARTMENTS.

ID	DNAME
BIOL	Biology
CS	Computer Science
MA	Mathematics

Table 2.4: Relation over CLASSES.

ID	CNAME	DEPT	PROFESSOR
MA121	Calculus I	MA	DoughertyM
MA200	Statistics	MA	CarrollC
CS230	Data Structures	CS	HayesD
MA386	Graph Theory	MA	CorrolC

Table 2.5: Relation over PROFESSORS.

ID	PNAME	DEPT
DoughertyM	Mara Dougherty	MA
HayesD	Darrel Hayes	CS
CarrollC	Carrie Carroll	MA

$\in \mathcal{D}$. The domain of SNAME is DOM(SNAME) = {'John', 'Jane', 'Emma', 'Tom'}.

A *relation schema* is a set of attributes describing the properties of the components of tuples. The set of attributes of a relation schema R is denoted schema(R). Let us look at Table 2.2 where we have a relation over STUDENTS, where schema(STUDENTS) = {ID, SNAME, AGE, ADDRESS, DEPT, DEGREE, YEAR}. We can do the same for Tables 2.3, 2.4, 2.5. A *database schema* is a finite set $\mathbf{R} = \{R_1, R_2, \ldots, R_n\}$ such that each $R_i \in \mathbf{R}$ is a relation schema. Let Tables 2.3, 2.4, 2.5 be a part of a database University, then our data schema is **University** = {Students, Departments, Classes, Professors}. We denote schema($\mathbf{R}$) = $\cup_{i \in I}$schema(R_i), where $I = \{1, 2, \ldots, n\}$.

2.3.2 Manipulative

The manipulative part of the relational model is where we define how to query and update our database. There are a couple of parts to the manipulative part, the relational algebra and the domain calculus. We will focus on the relational algebra which is a procedural query language for the relational model. The *relational algebra* is a collection of operators where each operator takes as input a single relation or a pair of relations and outputs a single relation. A *relational query* is a composition of a finite number of relational operators.

Definition 2.10. The union, $\cup$, of two relations r_1, r_2 over R is defined by

$$r_1 \cup r_2 = \{t \ : \ t \in r_1 \text{ or } t \in r_2\}$$

Definition 2.11. The difference, $-$, of two relations r_1, r_2 over R is defined by

$$r_1 - r_2 = \{t \ : \ t \in r_1 \text{ and } t \notin r_2\}$$

Definition 2.12. The projection, π, of a relation r over a relation schema R onto a set of attributes $Y \subseteq \text{schema}(R)$ is defined by

$$\pi_Y(R) = \{t[Y] \ : \ t \in r\}$$

Definition 2.13. Let r be a relation over relation schema R, A an attribute in schema(R) and B an attribute in $\mathcal{U}$, which is not in schema(R). The renaming, ρ, of A to B in r, is a relation over relation schema S, where schema(S) = (schema(R) $-$ $\{A\}$)$\cup\{B\}$, defined by

$$\rho_{A \to B}(r) = \{t \mid \exists u \in r \text{ such that } t[\text{schema}(R) - \{B\}] = u[\text{schema(R)} - \{A\}] \text{ and } t[B] = u[A]\}.$$

The union, difference, projection, and renaming are useful for basic needs, like needing to collect all the attributes of a relation or needing to find unique attributes, however, there are many times when we need more complex state-

ments to find what we are looking for. For example, in Table 2.5 if we want to find the classes that DoughertyM teaches, the projection would not help and the union and difference are also not useful operators. We need a selection of tuples from a relation r with respect to a selection formula F.

Definition 2.14. A *simple selection formula* over a schema R is either an expression of the form $A = a$ or an expression of the form $A = B$, where $A, B \in \text{schema}(R)$ and $a \in \text{DOM}(A)$.

A *selection formula* over R is a well-formed expression of one or more simple selection formulae over R together with the Boolean logical connectives, $\wedge$ (and), $\vee$ (or), $\neg$ (not) and parenthesis. A selection formula is called *positive* if it does not have any occurrence of $\neg$, and negative if it does.

Let r be a relation over a relation schema R, t a tuple in r, and F, F_1, F_2 selection formula over R. We say that t *logically implies* F, written $t \vDash F$, and is defined:

1. $t \vDash A = a$, if $t[A] = a$ evaluates to true
2. $t \vDash A = B$ if $t[A] = t[B]$ evaluates to true.
3. $t \vDash F_1 \wedge F_2$ if $t \vDash F_1$ evaluates to true and $t \vDash F_2$ evaluates to true.
4. $t \vDash F_1 \vee F_2$ if $t \vDash F_2$ evaluates to true or $t \vDash F_2$ evaluates to true.
5. $t \vDash \neg F$ if $t \vDash F$ does not evaluates to true.
6. $t \vDash (F)$ if $t \vDash F$.

Now we can define the selection operator which is used to filter data.

Definition 2.15. A *selection*, denoted σ, applied to a relation r over relation schema R with respect to a selection formula F over R, returns all tuples in r that logically imply F. We define σ by

$$\sigma_F(r) = \{t \ : \ t \in r \text{ and } t \vDash F\}$$

Now we can query "Retrieve the students who are studying math or whose address is Malet St", and express it as the selection $\sigma_F(r_1)$ where F is the formula DEPT = 'MA' $\vee$ ADDRESS = 'Malet St'. The below table is the result of our selection.

Table 2.6: Result of query $\sigma_F(r_1)$.

ID	SNAME	AGE	ADDRESS	DEPT	DEGREE	YEAR
500	John	20	Malet St	CS	BSC	first
623	Jane	22	Harold Rd	MA	BSC	fourth
522	Emma	18	Oxford St	MA	BSC	first

The *natural join* (or simply the join), $\bowtie$, of two relations r_1 over relation schema R_1 and r_2 over relation schema R_2 is a relation over relation schema R defined by

$$r_1 \bowtie r_2 = \{t \ : \ \exists t_1 \in r_1 \text{ and } \exists t_2 \in r_2 \text{ such that } t[\text{schema}(R_1)] = t_1 \text{ and } t[\text{schema}(R_2)] = t_2\},$$

where schema(R) = schema$(R_1) \cup$ schema(R_2). The *anti-join* is a join that

returns one copy of each tuple in the first relation that is not in the second.

2.3.3 Integrity

It is important to know how to restrict relations in order to satisfy certain conditions known as *integrity constraints* (or simply constraints). We view integrity constraints as first-order logic statements that restrict the set of allowable relations in a database. This is most commonly done by using primary keys and foreign keys.

Definition 2.16. An attribute or set of attributes whose values uniquely identify each entity in an instance of an entity type is called a *superkey* for the entity type. If E is an entity type then a *simple key* for E is a key for E which is composed of a single attribute.

In relational databases, there are two types of keys, primary keys and foreign keys. The *primary key* of a relation schema is a distinguished key designated by the database designer and is used as identification for each row of a table. So in Table 2.5 using the column ID as the primary key would be sufficient. Since primary keys are used for identification they cannot be NULL which means there cannot exist a row of data without a primary key, and primary keys cannot be repeated. A *foreign key* is a set of attributes in a relation schema that forms a primary key of another relation. For example in Table 2.4, PROFESSOR is a foreign key that references ID in Table 2.5. Primary keys are a special case of function dependencies, which generalize the

notion of keys and foreign keys are a special case of inclusion dependencies.

Chapter 3

Hypergraph Acyclicity

Unlike the one notion of acyclicity that exists for graphs, there are various degrees of acyclicity in hypergraphs. In increasing order of generality, we have three degrees of acyclicity, gamma acyclicity, beta acyclicity, and alpha acyclicity. We will pay particular interest to alpha acyclicity and beta acyclicity following the notation, definitions, and theorems from Brault-Baron's paper *Hypergraph Acyclicity Revisited* [4] unless otherwise specified.

First, we must discuss some preliminary definitions. Recall that a vertex v is incident with an edge e if $v \in e$. We will denote the set of incident edges of v as $I(v, H) := \{e \in H \mid v \in e\}$ [24], which is equivalent to the star of vertex v as stated in Brault-Baron's paper [4]. The *induced subhypergraph* H' on a set $S \subseteq V(H)$ of a hypergraph H, denoted $H' = H[S]$ is defined as $H[S] = \{e \cap S \mid e \in H\} \setminus \{\emptyset\}$. The *minimization* of a hypergraph H, denoted $M(H)$, is defined as $M(H) = \{e \in H : \nexists f \in H, e \subset f\}$. In other words,

the minimization of a hypergraph is the set of edges that are maximal for inclusion.

We call the set of vertices that are neighbors to x in H the *neighborhood* of x. A *clique* of a hypergraph is a nonempty subset of its vertices whose elements are pairwise neighbors. A hypergraph is *conformal* if each of its cliques of H is an edge in $M(H)$.

Recall that for a graph we define a cycle, which we will refer to as a *usual graph cycle*, as $P = (v_0, \ldots, v_k)$ where $v_0 = v_k$ and $k \geq 3$. We define a *cycle* on a hypergraph H as a tuple $\vec{t} = (t_1, \ldots, t_n)$ of n pairwise distinct vertices such that

$$M(H[\{t_1 : 1 \leq i \leq n\}]) = \{\{t_1, t_{i+1}\} : 1 \leq i < n\} \cup \{\{t_n, t_1\}\}.$$

A hypergraph H has a *cycle* if there exists a $\vec{t}$ that is a cycle of H. A hypergraph H is *cycle-free* if it has no cycle, and we will refer to this property as *cycle-freeness*. A graph is *acyclic* if it is cycle-free.

3.1 Alpha Acyclicity

Alpha acyclicity is the most general degree of acyclicity and the most studied. We will first provide a general definition of alpha acyclicity before defining some equivalent definitions and characterizations.

Definition 3.1. A hypergraph H is *alpha acyclic* if it is both conformal and

cycle-free.

Algorithm 1: GYO Algorithm.

Data: A hypergraph $H = (V, E)$
Result: Reduced hypergraph
1 **while** *not at end of* E **do**
2 **if** $e \in E$ *is contained in another hyperedge* e' **then**
3 delete e
4 **return** H
5 **while** *not at end of* E **do**
6 **if** $v \in V$ *is contained in exactly one hyerpedge* $e \in E$ **then**
7 delete v from e
8 **return** H
9 Continue lines 1-10 until H is empty or H cannot be reduced anymore

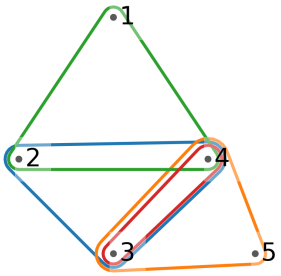
The above definition of alpha acyclicity is the more formal definition, however, when wanting to quickly determine if a hypergraph is alpha acyclic there are more efficient methods. In 1979 a generalization of perfect elimination orderings for alpha acyclic hypergraphs was given by Mark Graham [28]. The same result was also given independently by Yu and Özsoyoğlu in the same year [32]. This elimination ordering is known as Graham's reduction or the GYO algorithm. The GYO algorithm has two steps, the first step is to remove any hyperedges that are contained in another and the second step is to eliminate any vertices that exist in exactly one hyperedge. We continue this process until we get the empty hypergraph or we can no longer perform an elimination or reduction step. The GYO algorithm succeeds on a hypergraph H if the resulting hypergraph is the empty hypergraph.

Observe that for an alpha acyclic hypergraph H the neighborhood of a vertex $v \in e$, where $e \in E(H)$, and v is in no other hyperedge forms a clique since all vertices in $N(v)$ are pairwise neighbors. This means that any vertex that exists in only one hyperedge is a simplicial vertex. For hyperedges $e, f \in E(H)$ where $e \subseteq f$, every vertex in e would be a simplicial vertex since the neighborhood of each vertex would be contained in f. Interestingly the line graph of the dual of an alpha acyclic hypergraph H is chordal, where the dual H^* of H is obtained by interchanging the roles of the vertex-set and hyperedge-set [19]. Later in this section, we will prove that if the GYO algorithm succeeds then H is alpha acyclic as shown by Beeri et al. [2] and Goodman and Schmueli [20].

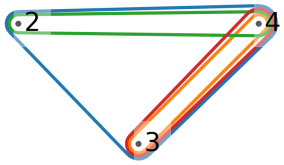
To show that an alpha acyclic hypergraph succeeds by the GYO algorithm we need to discuss a characterization first introduced by Brault-Baron [4].

Definition 3.2. A vertex x of a hypergraph H is an *alpha leaf* if $M(I(x, H))$ has a single edge; in other words $I(x, H)$ has a maximal element for inclusion. We say $\vec{v} = (v_1, \ldots, v_n)$ is an *alpha elimination order* of a hypergraph H if v_n is an alpha leaf and $(v_1, ..., v_{n-1})$ is an alpha elimination order of $H[V(H) \setminus \{x_n\}]$, which we will shorten to $H[\setminus\{x_n\}]$. The empty tuple is an alpha elimination order of the empty hypergraph. Observe that a vertex x is not an alpha leaf of H if and only if there are two vertices y and z such that $M(H[\{x, y, z\}]) = \{\{x, y\}, \{x, z\}\}$; in other words, $M(I(x, H)) = I(x, (M(H)))$ has two intersecting edges.

Example 3.1. In Figure 3.1a we can see that 1 and 5 are alpha leaves since



(a) Hypergraph H.



(b) Hypergraph $H[\backslash\{1,5\}]$

Figure 3.1: Alpha elimination ordering of a hypergraph H. This graph was generated using code in Appendix A.

$I(1,H) = \{\{1,2,4\}\} = M(I(x,H))$ and $I(5,H) = \{\{3,4,5\}\} = M(I(5,H))$ and 2, 3, 4 are not alpha leaves. Looking at vertex 3 we have $I(3,H) = \{\{3,4\},\{2,3,4\},\{3,4,5\}\}$ and $M(I(3,H)) = \{\{2,3,4\},\{3,4,5\}\}$. Note that if we take $H[\backslash\{1\}]$ then 5 is an alpha leaf of $H[\backslash\{1\}]$ and if we take $H[\backslash\{5\}]$ then 1 is still an alpha leaf of $H[\backslash\{5\}]$. Let $\vec{v} = (1,5)$ and $H[V(H)\setminus\{1,5\}] = \{\{2,3,4\},\{3,4\},\{2,4\},\{3,4\}\}$, Figure 3.1b. Observe that 2, 3, 4 are all alpha leaves since the edge $\{2,3,4\}$ is a maximal element for inclusion. Again it does not matter in what order we eliminate 2, 3, 4, since eliminating one does not affect whether the rest are alpha leaves. Thus one possible alpha elimination ordering is $\vec{v} = (1,5,2,3,4)$.

One distinct difference we can see between H and $H[\backslash\{1,5\}]$ in Figure 3.1 is the presence of a full hyperedge or a hyperedge that contains all vertices of a hypergraph, i.e. $V(H) \in H$. Recall that the GYO algorithm removes nested hyperedges, and in a hypergraph H with a full hyperedge, every other hyperedge is nested, which also means that for every vertex $v \in H$,

$M(I(v, H)) = V(H)$.

Lemma 3.1. *A hypergraph H with a full hyperedge, a hyperedge such that $V(H) \in H$, is alpha acyclic and every vertex is an alpha leaf.*

Recall a graph where every vertex has exactly two neighbors has a cycle. This leads us to the following theorem about alpha acyclic graphs.

Theorem 3.1. *An alpha acyclic nonempty hypergraph H such that $V(H) \notin H$ has two alpha leaves that are not neighbors.*

Proof. When a hypergraph H has size 1, $H = \{\{v\}\}$, the theorem is trivial since the premise is false. We will proceed by induction. Assume that the theorem holds for every hypergraph of size $k < n$ and let H be an alpha acyclic hypergraph of size n such that $V(H) \notin H$. Without loss of generality, we will assume that $H = M(H)$ because if $H \neq M(H)$ then $M(H)$ is an alpha acyclic hypergraph that is smaller than H without a full hyperedge. By induction, $M(H)$ must have two vertices that are alpha leaves of H. Therefore we will assume that $H = M(H)$.

Let H have two vertices. Then H is isomorphic to $\{\{x\}, \{y\}\}$. Therefore it satisfies our property. We will now assume that our hypergraph H has three or more vertices. We will assume that H has an alpha leaf, for a full proof of this fact see [4]. Now we will prove that H has two alpha leaves that are not neighbors.

Let x be our alpha leaf, e_x be the maximal edge of H that contains x, and $H' = H(\backslash\{x\})$. We have two cases $V(H') \in H'$ or $V(H') \notin H'$.

Case 1: Assume that $V(H') \in H'$. Because we assume that $H = M(H)$, we know that $e_x \neq V(H)$, so $V(H') \setminus e_x \neq \emptyset$. Let $y \in V(H') \setminus e_x$, then y is an alpha leaf of H, as shown in Lemma 3.1, that is not adjacent to x in H.
Case 2: Assume that $V(H') \notin H'$. Since we assume that H is alpha acyclic we we know that removing an alpha leaf means that H' is also alpha acyclic and we can find an alpha leaf. We continue by induction till we find two alpha leaves, y, z that are not neighbors in H'. Since $e_x \setminus \{x\} \neq V(H')$ then either y or z are not in $e_x \setminus \{x\}$. Without loss of generality suppose that y is not a neighbor of x in H, then $I(y, H) = I(y, H')$, thus y is an alpha leaf in H that is not a neighbor of x. □

Now we can prove that the GYO algorithm is an accurate tool in determining alpha acyclicity.

Theorem 3.2. *A hypergraph H is alpha acyclic if and only if H is GYO-reducible.*

Proof. We will proceed by induction on the size of a hypergraph H. Assume that the above holds for all hypergraphs of size less than n. Let H be of size n. Assume that H is alpha acyclic. Then H has an alpha elimination ordering. Let v be an alpha leaf of H. If v is a vertex that only appears in one hyperedge then we can remove the vertex and by induction $H[\setminus\{v\}]$ is GYO-reducible. If v appears in more than one hyperedge then v only appears in one hyperedge in $M(H)$. Therefore we can apply the nested edge removal rule and get H'. Since $M(H') = M(H)$, then H' is alpha acyclic because a

hypergraph is alpha acyclic if and only if $M(H)$ is.

Now suppose that H is GYO-reducible. If the first step of the GYO-reduction is a nested edge then the hypergraph after removal of the edge is alpha acyclic by the induction hypothesis, and since $M(H)$ is alpha acyclic then so is H. If the first step of the GYO-reduction is the removal of a vertex that appears in one hyperedge then by induction $H[\backslash\{x\}]$ is alpha acyclic and has an alpha elimination ordering. Since v is an alpha leaf, then there exists an alpha elimination order and H is alpha acyclic. □

It is important to understand alpha acyclicity because it is the most general meaning that every other degree of acyclicity will always imply alpha acyclicity. alpha acyclicity also plays an important role in database theory as we will see in Chapter 5.

3.2 Beta Acyclicity

Beta acyclicity is the second most general degree of acyclicity and has received less attention when compared to alpha acyclicity. However, alpha acyclic hypergraphs and beta acyclic hypergraphs share a strong connection as shown in the definition of beta acyclic.

Definition 3.3. A hypergraph H is *beta acyclic* if all of its subhypergraphs are alpha acyclic. Equivalently H is beta acyclic if and only if every subhypergraph is cycle-free.

Alternatively we can define a *beta-cycle* in a simple hypergraph as a cycle:

$$(x_0, e_1, x_1, e_2, x_2, \ldots, e_k, x_k = x_0), k \geq 3$$

such that for all $i \in \{0, 1, 2, \ldots, k-1\}$, x_i belongs to e_i and e_{i+1} and no other e_j from the cycle [24]. A hypergraph is beta acyclic if it does not contain a beta-cycle.

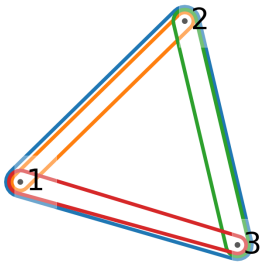


Figure 3.2: This is an example of an alpha acyclic hypergraph. Note that this is not beta acyclic because $(v_1, e_2, v_2, e_3, v_3, e_4, v_1)$ is a beta-cycle.

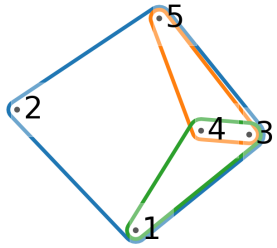


Figure 3.3: This is a hypergraph generated by code in Appendix A and is a beta acyclic hypergraph. Note that this hypergraph is also alpha acyclic since the GYO algorithm succeeds.

Observe that for a hypergraph H that is beta acyclic, any subhypergraph H' is also beta acyclic. If all subhypergraphs of H are alpha acyclic then all subhypergraphs of H' must also be alpha acyclic. We do not get the hereditary property with alpha acyclic hypergraphs. Since any hypergraph with a full hyperedge is alpha acyclic, there can exist a subhypergraph without a full hyperedge that is not alpha acyclic.

Recall that an alpha acyclic hypergraph will have an alpha-elimination

ordering and at least two alpha leaves. We can extend these notions to beta acyclic hypergraphs, but since not every alpha acyclic hypergraph is beta acyclic we need further restrictions on our beta leaf. Note that a beta leaf is more commonly called a nest-point which is the terminology that we will use, deviating from Brault-Baron's paper [4] in language, while using Brault-Baron's definitions and theorems.

Definition 3.4. A vertex v of a hypergraph H is a *nest-point* if for all $e, f \in I(v, H)$, $e \subseteq f$ or $f \subseteq e$. We say $\vec{v} = (v_1, \ldots, v_n)$ is a *nest-point elimination order* of a hypergraph H if v_n is a nest-point and $(v_1, \ldots, v_{n-1})$ is a nest-point elimination order of $H[\backslash\{x_n\}]$.

Let us look at a nest-point elimination ordering for our hypergraph H in Figure 3.3. Observe that every single vertex is an alpha leaf, but only $1, 2, 5$ are nest-points. To see why vertices 3 and 4 are not nest-points let us look at vertex 4. We have $I(4, H) = \{\{1, 2, 4, 3\}, \{3, 4, 5\}, \{1, 3, 4\}\}$ and $\{3, 4, 5\} \not\subseteq \{1, 3, 4\}$ and $\{1, 3, 4\} \not\subseteq \{3, 4, 5\}$, so $I(4, H)$ is not linearly ordered. Similar to an alpha elimination ordering, the order in which we remove nest-points does not matter. Let $\vec{v} = (1, 2, 5)$, then we have the hypergraph $H[\backslash\{1, 2, 5\}] = (\{3, 4\}, \{\{3, 4\}, \{3, 4\}\})$. We will ignore repeated hyperedges since they do not change the outcome. We can see that both $3, 4$ are nest-points, so one nest-point elimination ordering is $\vec{v} = (1, 2, 5, 3, 4)$.

We can apply Lemma 3.1 and Theorem 3.1 to nest-point elimination orders, by replacing alpha leaves with nest-points.

Theorem 3.3. *A beta acyclic hypergraph H with at least two vertices has two nest-points that are not neighbors in $H \setminus \{V(H)\}$.*

Proof. We will use induction on the size of a hypergraph H. Assume that the result holds for every hypergraph of size n. Let H be beta acyclic and of size n where $n \geq 2$. If $H = \{V(H)\}$ the result is trivial, so assume $H \neq \{V(H)\}$. Case 1: Let $V(H) \in H$ and $H' = H \setminus \{V(H)\}$. Recall that beta acyclicity is hereditary, so H' is beta acyclic and contains at least one vertex. If $V(H) = V(H')$ then by induction H' has two nest-points that are not neighbors in $H' \setminus \{V(H')\} = H \setminus \{V(H)\}$ and are nest-points of H. If $V(H) \neq V(H')$, there exists a vertex $v \in V(H) \setminus V(H')$ that only belongs to the edge $V(H) \in H$, and is thus a nest-point. Since H' is also beta acyclic, there exists a beta leaf y that is also a beta leaf in H. Thus in both cases there exists two beta leaves in H that are not neighbors.
Case 2: Let $V(H) \notin H$. If H is isomorphic to $\{\{x\}, \{y\}\}$ then we satisfy our result, so let us assume this is not the case and H has three or more vertices. First we will prove the following fact:

Fact: If H has an alpha leaf x then H has a nest-point that is not a neighbor of x in $H \setminus \{V(H)\}$.

Proof of Fact. Let $H' = H[\setminus\{x\}]$. Then $e_v \setminus \{x\} \neq V(H[\setminus\{x\}])$ since $e_x \neq V(H)$. We know H' is beta acyclic, so by induction it has two beta leaves that are not neighbors in $H' \setminus V(H')$. More specifically there exists vertices y, z that are not neighbors in the edge $e_x \setminus \{x\}$ in

H'. Without loss of generality let $y \notin e_x$, so y is not a neighbor of x in H. Therefore $I(y, H) = I(y, H')$, and since y is a nest-point of H' it must be a nest-point of H and is not a neighbor of x. Therefore $y \notin H \setminus \{V(H)\}$. □

We know that H is alpha acyclic because beta acyclicity implies alpha acyclicity, so there exists an alpha leaf. Therefore we can find a nest-point y. We know that a nest-point is an alpha leaf, so we can apply the fact again to y and get a vertex z that is also a nest-point such that y and z are not neighbors in H. Hence we there are at least two nest-points that are not neighbors in $H \setminus \{V(H)\}$ for a beta acyclic hypergraph with at least two vertices. □

Recall the GYO algorithm and its relation to perfect elimination orderings. From this we were able to infer that alpha acyclic hypergraphs have a correlation with chordal graphs. Since every subhypergraph of a beta acyclic hypergraph is alpha acyclic, one would naturally wonder if there is an association between chordal graphs and beta acyclic hypergraphs. Beta acyclicity is a stricter degree of acyclicity than alpha acyclicity, and strongly chordal graphs are stricter than chordal graphs, so we have a natural relation between beta acyclic hypergraphs and strongly chordal graphs. Brouwer and Kolen [7] showed that a hypergraph H is beta acyclic if and only if every induced subhypergraph has a nest-point. Recall that if a graph is strongly chordal we can find a simple elimination ordering and a simple vertex is a vertex whose closed neighborhood is linearly ordered. A nest-point is a

vertex whose incident edges are linearly ordered, where incident edges are the sets of vertices that are neighbors. So, the existence of nest-points in beta acyclic hypergraphs corresponds to the existence of simple vertices in strongly chordal graphs.

We will now introduce a family of hypergraphs that are always beta acyclic. A hypergraph H is *totally balanced* if every cycle $C = x_0, e_1, x_1, e_2, x_2, \ldots, x_k, e_k, x_{k+1} = x$, with a length ≥ 3 has a hyperedge e_i of the cycle which contains at least three vertices of the cycle [19]. Recall that a beta-cycle is a cycle of length three or more such that a vertex belongs to only two hyperedges, in other words a hyperedge only contains two vertices of the cycle. This means that totally balanced graphs do no contain any beta-cycles and thus are always beta acyclic. Similarly we can say that every beta acyclic hypergraph is totally balanced which is a theorem of D'Atri and Moscarini [11]. This tells us that every subhypergraph of a totally balanced hypergraph is totally balanced.

Theorem 3.4 ([6])**.** *Every totally balanced (beta acyclic) hypergraph has a line graph that is chordal.*

Proof. Let H be a totally balanced hypergraph with cycle

$$C = x_1, e_1, x_2, e_2, x_3, \ldots, x_k, e_k, x_{k+1}.$$

This cycle correlates to a cycle in the line graph, so they both have the same length. Because H is totally balanced C has a hyperedge of the cycle that

contains at least three vertices of the cycle. This correlates to a chord in our line graph. □

Because every subhypergraph of a beta acyclic hypergraphs is beta acyclic and alpha acyclic, beta acyclicity becomes a desirable property for hypergraphs to have. This has led to various generalizations of beta acyclicity, however many of them do not retain every desirable property. Notably some generalizations cannot be solved in a reasonable time and others do not retain tractability of conjunctive queries with negation. In the next chapter we will discuss a generalization of beta acyclicity which uses a new notion of hypergraph width to generalize the existence of nest-point elimination orders.

Chapter 4

Nest-Set Elimination Ordering

Generalizing beta acyclicity allows for more hypergraphs to reap the benefits of a beta acyclic hypergraph structure. Work by Gottlob and Pitcher [22] has introduced the notion of beta hypertree width, an analogous notion to alpha hypertree width which is a generalization of alpha acyclicity [21]. However, unlike alpha hypertree width, beta hypertree width does not preserve tractability when used in practice. In more recent work Carbonell et al. [8] introduced point-width, however, point-width cannot be determined in polynomial time, while beta acyclicity can. Inspired by these results, Lanzinger introduced a novel generalization of beta acyclicity called nest-set width which extends the notion of nest-points and nest-point elimination orderings. [24].

4.1 Nest-Set Width

Throughout this section we will follow the definitions, lemmas, corollaries, and notation of Lanzinger's paper [24] unless otherwise specified. Recall the set of incident edges of vertex v is defined as $I(v, H) := \{e \in H \mid v \in e\}$. We can extend $I(v, H)$ to a set of vertices $s = \{v_1, \ldots, v_\ell\}$ where $I(s, H) := \cup_{i=1}^{\ell} I(v_i, H)$. We say that for some $e \in I(s, H)$, e is incident to the set s and if context makes it clear what H is then we simplify $I(s, H)$ to $I(s)$.

First we extend the notion of nest-points to nest-sets.

Definition 4.1. Let H be a hypergraph. A non-empty set $s \subseteq V(H)$ of vertices is called a *nest-set* in H if the set

$$I^*(s, H) := \{e \backslash s \mid e \in I(s, H)\}$$

is linearly ordered by inclusion.

Note that every nest-point is a nest-set with one element, so we can also extend the notion of nest-point elimination orderings to get the nest-set elimination ordering.

Definition 4.2. Let H be a hypergraph and let $\mathcal{O} = (s_1, \ldots, s_q)$ be a sequence of sets of vertices. Define $H_0 = H$ and $H_i := H_{i-1} \setminus s_i$. We call $\mathcal{O}$ a *nest-set elimination ordering* (NEO) if, for each $i \in [q]$, s_i is a nest-set of H_{i-1} and H_q is the empty hypergraph.

A nest-set s with at most k elements is called a k-nest-set and a nest-set elimination ordering that consists of only k-nest-sets is a k-nest-set elimination ordering or k-NEO.

Definition 4.3. Given a hypergraph H, the nest-set width of H, $nsw(H)$, is the lowest k for which there exists a k-NEO.

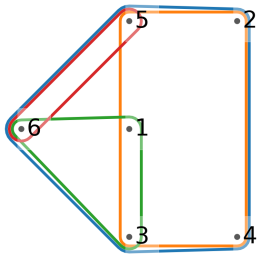
Corollary 4.1. *A hypergraph H has a $nsw(H) = 1$ if and only if H is β-acyclic.*

To understand nest-set elimination ordering and nest-sets let us look at a hypergraph with a 2-NEO. Let

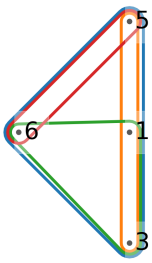
$$H(V, E) = (\{1, 2, 3, 4, 5, 6\}, \{\{1, 2, 3, 4, 5, 6\}, \{1, 2, 3, 4, 5\}, \{1, 3, 6\}, \{5, 6\}\})$$

be a hypergraph with 6 vertices and 4 hyperedges, Figure 4.1a. To find a 2-NEO we need to first find a 2-nest-set. Let $H_0 = H$ and $s_1 = \{2, 4\}$. We need to verify that s_1 is a 2-nest-set. Observe that $I(s_1, H_0) = \{\{1, 2, 3, 4, 5, 6\}, \{1, 2, 3, 4, 5\}\}$ and $I^*(s_1, H_0) = \{\{1, 3, 5, 6\}, \{1, 3, 5\}\}$ is linearly ordered. Thus, s_1 is a 2-nest-set of H_0 which means we can proceed with our elimination ordering. Now we must find a 2-nest-set for $H_1 = H_0 \backslash s_1$, shown in Figure 4.1b. Let $s_2 = \{5, 6\}$. We need to verify that s_2 is a nest-set, $I(s_2, H_1) = \{\{1, 3, 5, 6\}, \{1, 3, 5\}, \{1, 3, 6\}, \{5, 6\}\}$ and $I^*(s_2, H_1) = \{\{1, 3\}, \{1, 3\}, \{1, 3\}, \emptyset\}$, so $I^*(s_2, H_1)$ is linearly ordered. Therefore s_2 is a nest-set and we have $H_2 = H_1 \setminus s_2$, shown in Figure 4.1c. Since we are left with a hypergraph

with one edge and two vertices, we have our final 2-nest-set, $s_3 = \{1, 3\}$ where $I(s_3, H_2) = \{\{1, 3\}\}$ and $I^*(s_3, H_2) = \{\emptyset\}$ which is linearly ordered. Hence we have the 2-NEO $\mathcal{O} = (\{2, 4\}, \{5, 6\}, \{1, 3\})$. Observe that H has no 1-NEO, so $nsw(H) = 2$.



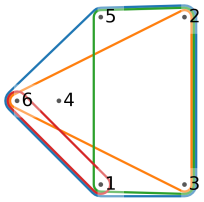
(a) Original hypergraph, H_0.



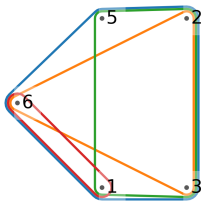
(b) $H_1 = H_0 \setminus \{2, 4\}$.

(c) $H_2 = H_1 \setminus \{5, 6\}$.

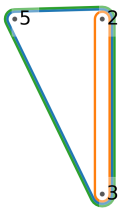
Figure 4.1: Example of a 2-nest-set elimination ordering of a hypergraph H generated by code in Appendix A.



(a) Original hypergraph, H_0.



(b) $H_1 = H_0 \setminus \{4\}$.



(c) $H_2 = H_1 \setminus \{1, 6\}$.

(d) $H_3 = H_2 \setminus \{2, 3\}$.

Figure 4.2: Example of a 2-nest-set elimination ordering with different size nest-sets of a hypergraph H generated by code in Appendix A.

Let us look at a second example where not all of the nest-sets have strictly two elements in them. Note that a 1-nest-set is a 2-nest-set, 3-nest-set, 4-nest-set, etc. but a 2-nest-set is not necessarily made up of 1-nest-sets. Let $H_0 = (\{1, 2, 3, 4, 5, 6\}, \{\{1, 2, 3, 4, 5, 6\}, \{1, 2, 3, 5\}, \{2, 3, 4, 6\}, \{1, 6\}\})$, Figure 4.2a.

Note that H_0 has no 2-nest-sets that have exactly 2 elements, but it does have a 2-nest-set with one element, $\{4\}$. So we get the following 2-NEO: $(\{4\}, \{1,6\}, \{2,3\}, \{5\})$.

Beta acyclic hypergraphs are hereditary, which means every subhypergraph of a beta acyclic hypergraph is beta acyclic. Having properties that are hereditary are desirable because it means that any induced subhypergraph will inherit that property. Because of the close connection between nest-point elimination orderings and nest-set elimination orderings, one would infer that nest-set elimination orderings are also hereditary. Let us look at a motivating example to see if we can glean any insight. One of the smallest hypergraphs without a 2-NEO is the square, Figure 4.3, which is conformal, but not cycle-free. We claim that any hypergraph with H_s as a subhypergraph will not have a 2-NEO.

Example 4.1.

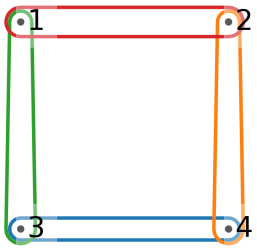


Figure 4.3: The square hypergraph, H_s.

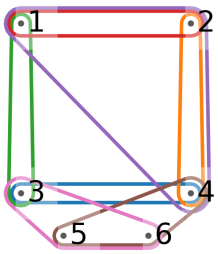


Figure 4.4: A hypergraph H where $H_s \subset H$.

In Figure 4.4, H is a hypergraph with 6 vertices and 7 hyperedges. Observe that $H_s = (\{1,2,3,4\}, \{\{1,2\}, \{1,3\}, \{2,4\}, \{3,4\}\})$ is a subhypergraph of H

and H does not have 2-NEO.

Lemma 4.1 (Motivation for [24])**.** *If H_s is a subhypergraph of a hypergraph H, then H does not have a 2-NEO.*

Proof. Let H be a hypergraph where H_s is a subhypergraph. Suppose on the contrary that H has a 2-NEO. Then there exists a sequence of sets of vertices $\mathcal{O} = (s_1, s_2, \ldots, s_q)$ such that each s_i is a 2-nest-set of H_i and H_q is the empty hypergraph. Note that the order in which we eliminate nest-sets does not matter. Because $\mathcal{O}$ is a 2-NEO there exists at most 4 nest-sets containing vertices from H_s. Let $H_s = \{\{v_1, v_2, v_3, v_4\}, \{\{v_1, v_2\}, \{v_1, v_3\}, \{v_2, v_4\}, \{v_3, v_4\}\}\}$. Without loss of generality suppose that $v_1 \in s_i$ where s_i is a 2-nest-set.

Case 1: Suppose that s_i is the first nest-set containing a vertex from H_s in $\mathcal{O}$. Then $|s_i| = 1$ or $|s_i| = 2$. If $|s_i| = 1$ then $s_i = \{v_1\}$ and $I^*(s_i, H)$ is linearly ordered. However this is not possible since $I^*(s_i, H) = \{\{v_2\}, \{v_3\}, C \setminus \{v_1\}\}$, where C is the set of edges other than $\{v_1, v_2\}$ and $\{v_1, v_3\}$ that v_1 is incident to. If $|s_i| = 2$ then $s_i = \{v_1, v\}$ where $v \in H$ and $v \notin H_s$. Observe that we run into the same problem with v_2 and v_3. In both cases we contradict the fact that s_i is a nest-set.

Case 2: Suppose that s_i is not the first 2-nest-set containing a vertex from H_s. If there exists a 2-nest-set with v_4, then we have Case 1 since v_4 and v_1 are not neighbors. Without loss of generality let s_j be a 2-nest-set containing v_2, then $I^*(s_i, H) = \{\{v_3\}, C \setminus \{s_i\}\}$ is linearly ordered where C is the set of edges that s_i is incident to. However, this contradicts the fact that H_s is a

subhypergraph and $v_1, v_2 \in H_s$ and are neighbors. For $v_2 \in s_j$, s_j cannot be linearly ordered as shown in Case 1, so s_j is not a 2-nest-set.

Therefore if $\mathcal{O}$ exists then $H_s \nsubseteq H$. Hence, a hypergraph H with subhypergraph H_s does not have a 2-NEO. □

This lemma leads us to a more general conclusion highlighted in the following two lemmas. First we will establish that a hypergraph with a k-NEO remains valid after removing a vertex and then show that this also applies for removing hyperedges.

Lemma 4.2. *Let H be a hypergraph with k-NEO, $\mathcal{O} = (s_1, s_2, \ldots, s_\ell)$ and $r \subseteq V(H)$. Then the sequence $\mathcal{O}' = (s_1 \setminus r, s_2 \setminus r, \ldots, s_\ell \setminus r)$ is a k-NEO of $H \setminus r$. Note that this may result in some s_i being empty and nest-sets are non-empty. If this occurs then we remove the empty set from the NEO.*

Proof. We want to show that for any nest-set $s \in \mathcal{O}$, $s \setminus r$ is a nest-set of $H \setminus r$ or the empty set. Suppose on the contrary that $s \setminus r$ is not the empty set and not a nest-set of $H \setminus r$. Then there exists two hyperedges $e_1, e_2 \in I(s \setminus r, H \setminus r)$ such that $e_1 \nsubseteq e_2$ and $e_2 \nsubseteq e_1$. We know there exists some $e_1', e_2' \in I(s, H)$ such that $e_1 = e_1' \setminus r$ and $e_2 = e_2' \setminus r$ and $e_1' \subseteq e_2'$ or $e_2' \subseteq e_1'$. Without loss of generality this must mean $e_1' \setminus r \subseteq e_2' \setminus r$, contradicting our assumption that $I^*(s \setminus r, H \setminus r)$ is not a nest-set.

It follows that $s_1 \setminus r$ is a k-nest-set of $H \setminus r$. To show that $\mathcal{O}'$ is a k-nest-set elimination ordering of $H \setminus r$, we need to show that s_2 is a k-nest-set of $(H \setminus r) \setminus s_1$, however the same observation from the beginning of our proof

applies. We can then repeat this argument for all s_i until s_ℓ. Thus $\mathcal{O}$ is a k-NEO. □

Let us look at our hypergraph H_0 from Figure 4.1. Let $r = \{1, 4\}$. Then $\mathcal{O}' = (\{2\}, \{5, 6\}, \{3\})$ and $H \setminus r = (\{2, 3, 5, 6\}, \{2, 3, 5\}, \{\{2, 3, 5\}, \{3, 6\}, \{5, 6\}\})$. It is easy to confirm that each s_i is a nest-set and that $\mathcal{O}'$ is a nest-set elimination ordering.

From Lemma 4.2 we can conclude that k-NEOs are hereditary, leading to the following lemma.

Lemma 4.3. *Let H be a hypergraph with k-NEO, $\mathcal{O} = (s_1, \ldots, s_\ell)$. Let H' be a connected subgraph of H and $\Delta = V(H) \setminus V(H')$ the set of vertices no longer present in the subhypergraph. Then the sequence $(s_1 \setminus \Delta, \ldots, s_\ell \setminus \Delta)$ si a k-NEO of H'.*

Proof. From the proof of Lemma 4.2 we know that for any $s \in \mathcal{O}$, $s \setminus \Delta$ is going to be a nest-set of $H \setminus \Delta$ or the empty set. Thus $I^*(s \setminus \Delta, H \setminus \Delta)$ is linearly ordered and H' is a subhypergraph of H and does not contain any vertices from Δ, we have

$$E(H') = E(H' \setminus \Delta) \subseteq E(H \setminus \Delta).$$

Thus $I^*(s \setminus \Delta, H') \subseteq I^*(s \setminus \Delta, H \setminus \Delta)$, so $I^*(s \setminus \Delta, H')$ is linearly ordered and $s \setminus \Delta$ is a nest-set. We can again use the same argument across the nest-set elimination ordering like in Lemma 4.2 to prove the statement. □

Our aim in discussing nest-set width is to determine if it can be used as a suitable generalization for beta acyclicity. Specifically, we are interested in whether nsw can help maintain the tractability of queries in relational databases. However, if we can't identify nsw in a reasonable manner, we can't benefit from its tractability. For a constant k we can find a k-NEO in polynomial time by checking all combinations of vertices up to k vertices to see if any are a nest-set. To get the nest-set elimination ordering we can take the first found nest-set and take the hypergraph with the nest-set removed and continue the process till no nest-sets are found or we get the empty hypergraph. This is considered a greedy approach.

Recall that nest-set width is not the same as the nest-set elimination ordering, a beta acyclic hypergraph has a k-NEO for any $k \geq 1$, but has a nsw of 1. Since we cannot fix k when looking for the nsw of a hypergraph this leads to a problem since nsw cannot be found in polynomial time. Finding whether a hypergraph H has a k-nest-set is NP-complete and as a consequence, nsw is NP-complete. However, when parameterized by k, nest-set width is fixed-parameter tractable which means it can be solved in time $f(k) \cdot |n|^{O(1)}$, where f is a function of k independent from n. The proof of these results as well as an algorithm to find a k-nest-set where k is not constant, can be found in Lanzinger's paper [24].

4.2 Nest-Set Elimination Observations

Nest-set width is a generalization of beta acyclicity which leaves questions about which properties of beta acyclicity carry over to nest-set width and nest-set elimination orderings. To better understand nest-sets and nest-set elimination orderings we need to evaluate various different hypergraphs. Generating hypergraphs by hand is tedious and leaves a lot of room for error, and the potential of generating isomorphic hypergraphs. However, because of the usefulness of graphs in computer science there are ways to represent both graphs and hypergraphs for computational use. We are able to randomly generate hypergraphs based on the number of vertices, number of hyperedges, and the size of each hyperedge, Appendix A. However, while generating hypergraphs is not too computationally complex, checking if hypergraphs are isomorphic is, leading to long run times [27]. Because of this and the exponential nature of the number of hypergraphs that exist as the number of vertices increases, hypergraphs of small size, ≤ 6 vertices, are the main focus.

Each randomly generated hypergraph was tested for alpha acyclicity through use of the GYO algorithm, Appendix B, beta acyclicity through the use nest-point elimination orderings, Appendix C, and existence of a 2-NEO through the use of nest-set elimination orderings, Appendix D. Additionally each graph can be plotted for a visual representation. The hypergraphs, their generating values, and their truth-value of conditions, true if alpha acyclic,

false if not etc., were stored into a database, see Appendix E.

In this section we will discuss nest-set elimination ordering in more depth and present various observations. We will specifically look at 2-NEOs and always assume that our hypergraphs are connected and contain no repeated hyperedges. We only consider hypergraphs with no repeated hyperedges because repeating hyperedges do not affect a 2-NEO existing as proven in Lemma 4.3. All results in this section were gleaned from evaluating the database of generated hypergraphs as described above.

Observe that a hypergraph with at most two hyperedges and at most 2 vertices will always be beta acyclic. However, this does not hold true for hypergraphs with three vertices. Because nest-set elimination orderings generalize beta acyclicity k-NEOs naturally capture a larger amount of hypergraphs. This leads us to our first observation about hypergraphs and 2-NEOs.

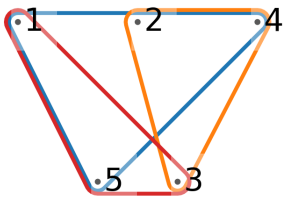
Theorem 4.1. *Every connected hypergraph with 3 vertices and no repeated hyperedges has a 2-NEO.*

Proof. Let H_0 be a hypergraph that satisfies our conditions above and $V(H_0) = \{v_1, v_2, v_3\}$. If there exists a 2-nest-set then $|E(H_1)| = 1$ or $|E(H_1)| = 2$, so H_1 is beta acyclic. Therefore we only need to prove the existence of of a 2-nest-set in a hypergraph with 3 vertices. Let us assume on the contrary that no 2-nest-set exists, then there exists no set of at most two vertices, s, such that $I^*(s, H_0)$ is linearly ordered. Without loss of generality let $s = \{v_1, v_2\}$. Then $I^*(s_1, H_0)$ only has one element, namely $\{v_3\}$. Note that any set with

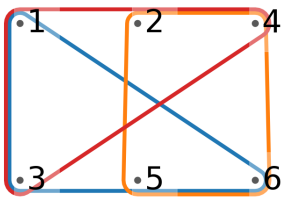
one element is linearly ordered by inclusion. So any set with two vertices is a nest-set leading to a contradiction. This means that there always exists a 2-nest-set when given a connected hypergraph with 3 vertices.

□

One may conjecture that a hypergraph with 3 hyperedges would also have a 2-NEO, however this only holds for hypergraphs with ≤ 4 vertices.



(a) Hypergraph with 5 vertices 3 hyperedges and no 2-NEO.



(b) Hypergraph with 6 vertices 3 hyperedges and no 2-NEO.

Figure 4.5: Hypergraphs with 3 hyperedges and no 2-NEO, generated by code from Appendix A. Observe that neither of these hypergraphs have a 2-nest-set because any combination will either have a singleton that is not nested in every hyperedge, or two hyperedges of the same size that are not equal.

It is important to note that we are discussing nest-set elimination ordering in Theorem 4.1, not nest-set width which is the minimum k-NEO of a hypergraph. We can omit the case with a nest-set with one element because we are guaranteed to have a nest-set with two elements in a connected hypergraph with three vertices, but are not guaranteed a nest-set with one

element. Therefore a hypergraph with 3 vertices will have a nsw at most 2, and we can apply a similar argument to hypergraphs with 4 vertices.

Corollary 4.2. *A connected hypergraph with 4 vertices and no repeated edges will always have a 2-NEO if at least one 2-nest-set exists. In other words the only way for a hypergraph with 4 vertices to not have a 2-NEO is for no 2-nest-set to exist.*

There is a strong connection between beta acyclic hypergraphs and alpha acyclic hypergrphs. Our goal in defining and discussing properties of nest-set width is to generalize beta acyclicity to retain tractability, so it would be natural to think that there would be a connection between nsw and alpha acyclicity.

However, since not every hypergraph with a nsw is beta acyclic, or even alpha acyclic, Figure 4.6, any connection between nsw and alpha acyclicity is not obvious or as general as it is for beta acyclicity. Instead we will specifically look at 2-NEOs and its connection with alpha acyclicity. First let us examine the following lemma.

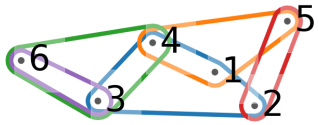


Figure 4.6: Hypergraph with a 2-NEO, but not alpha acyclic [24].

Proposition 4.1. *Let H be a connected hypergraph with 4 vertices. If $|H| \leq 3$, H has three or fewer hyperedges, then H has a 2-NEO.*

Proof. From Corollary 4.2 it is sufficient to show that there exists a 2-nest-set. Let e_v represent a full hyperedge, a hyperedge that contains all vertices

of H.

Case 1: Suppose that $e_v \in E(H)$. Let $H' = H \setminus e_v$. Then $|E(H')| \leq 2$ and we know hypergraphs with at most 2 hyperedges are beta acyclic. Therefore H' is beta acyclic, and since e_v is a full edge all other hyperedges are nested in it, so it does not affect our linear order. Thus every hypergraph with 3 vertices and a full edge is beta acyclic, which means there exists a 2-NEO.

Case 2: Suppose that $e_v \notin H$. If $|E(H)| = 2$ we know H is beta acyclic, so there exists a 2-NEO. Let us assume that $|E(H)| = 3$. If H has no hyperedge with 3 vertices then there must exist two nest-points because we assume H is connected, so H would have a 2-NEO. Assume there exists a hyperedge $e \in H$ such that $|e| = 3$. If there exists a hyperedge $e_1 \in H$ such that $|e_1| = 2$ and $e_1 \subset e$, then there exists a $v \in V(H)$, such that $v \notin e$ and $v \notin e_1$. Thus v is a nest-point because it exists in only one hyperedge. Suppose that no such e_1 exists. Then there exists a $x, y \in V(H)$ such that $x \in e_{xy}, e_x$ and $y \in e_{xy}, e_y$ where $e_{xy}, e_x, e_y \in E(H)$ and $e_x \cap e_y \neq \emptyset$. We claim that $\{x, y\}$ is a nest-set.

Let $s = \{x, y\}$. Then $I^*(s, H) = \{e_{xy} \setminus s, e_x \setminus s, e_y \setminus s\}$. We want $I^*(s, H)$ to be linearly ordered. Assume on the contrary $I^*(s, H)$ is not linearly ordered. Since we are working with a hypergraph with 4 vertices, elimination of s leaves us with two vertices. Thus for $I^*(s, H)$ to not be linearly ordered there must exist two other vertices, $w, z \in V(H)$ such that $\{w\}, \{z\} \in I^*(s, H)$. However for this to be true, we either contradict our assumption that x, y share a hyperedge or contradict the assumption that at least one hyperedge

has 3 vertices. Therefore there exists at least one 2-nest-set in H, getting us our result.

□

Now that we know every hypergraph H with 4 vertices and $|E(H)| \leq 3$ has a 2-NEO we will show the following proposition.

Proposition 4.2. *Let H be a connected hypergraph with 4 vertices. If $|E(H)| = 4$ then for every hypergraph that does not have a 2-nest-set $H \setminus \{V(H)\}$ is not α-acyclic.*

Proof. Let H be a hypergraph with 4 vertices and 4 hyperedges. Suppose $e_v \in E(H)$, where e_v is a full hyperedge and $H' = H \setminus e_v$. If H' is isomorphic to a hypergraph with 4 vertices and 3 hyperedges then there exists a 2-NEO because we are removing a full hyperedge. Let us assume that H' is not isomorphic to a hypergraph with 4 vertices and 3 hyperedges. Then H' must be disconnected, so there exists a $v \in H$ where $v \notin H'$, so v is a nest-point.

Let us now assume that $e_v \notin E(H)$. Suppose that $H = M(H)$. Then H has non nested hyperedges. Let us assume that H does not have a 2-NEO, so no 2-nest-set exists. Observe that this means that the GYO algorithm cannot fully reduce H since no vertex exists in only one hyperedge and there are no nested hyperedges by assumption.

Now suppose that $H \neq M(H)$. Assume that H does not have a 2-NEO, and consequently no 2-nest-set. Then there exists no two vertices $x, y \in V(H)$ such that $I^*(\{x, y\}, H)$ is linearly ordered. However this contradicts

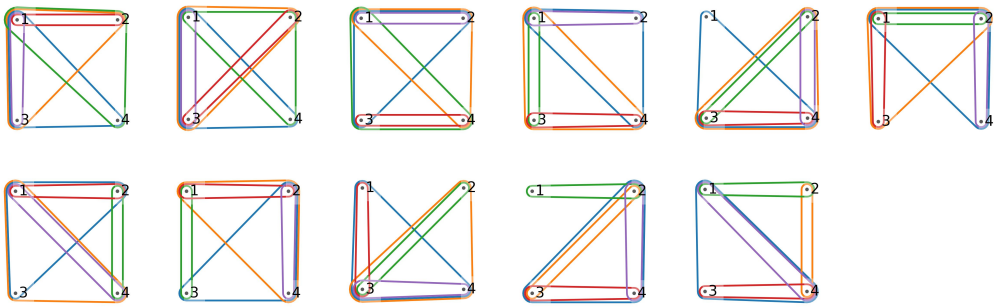


Figure 4.7: Hypergraphs with 4 vertices and 4 hyperedges that have a 2-NEO. None of these hypergraphs are isomorphic and they were all randomly generated based off of code from Appendix A.

the fact there are are nested hyperedges. Let $V(H) = \{x, y, w, z\}$. Without loss of generality suppose $\{x, y\}$ is a nested hyperedge. If $I^*(\{x, y\}, H)$ is not linearly ordered then $\{w\}, \{z\} \in I^*(\{x, y\}, H)$. However this can only be true if there is a vertex that exists in only one hyperedge which we know is a nest-point and therefore a 2-nest-set contradicting the fact that there is no 2-NEO.

□

We can extend the above proposition to hypergraphs with 4 vertices and 5 hyperedge, but will take a different approach to proving it. We will instead assume that H has no 2-NEO and try to construct an alpha acyclic hypergraph.

Proposition 4.3. *Let H be a connected hypergraph with 4 vertices. If $|E(H)| = 5$ then for every hypergraph that does not have a 2-nest-set $H \setminus \{V(H)\}$ is not α-acyclic.*

Proof. Let $|E(H)| = 5$. If $e_v \in E(H)$ then $H \setminus e_v$ is isomorphic to a hypergraph with 4 vertices and 4 hyperedges, so our result holds. Let us assume that $e_v \notin H$. Observe that there exists one hypergraph such that $H = M(H)$, namely the hypergraph where all 5 hyperedges have 2 vertices and $H_s \subseteq H$ where H_s is the square hypergraph. By Lemma 4.1 we know that this means that there is no 2-NEO, and since there are no vertices that exist in one hyperedge and no nested hyperedges it is not alpha acyclic.

Now let us assume that $H \neq M(H)$. Assume on the contrary that H is alpha acyclic, we want to show that there must exist a 2-nest-set. We will proceed using the GYO algorithm referring to the two steps as eliminate, there is a vertex that exists in only one hyperedge, and reduce, there is a nested hyperedge, and go step by step.

Step 1: If we can eliminate then there exists a nest-point contradicting the fact that there is no 2-nest-set, so we can only reduce. Let $e_1 \subset e_2$ where $e_1, e_2 \in E(H)$.

Step 2: If we can eliminate then $v \in V(H)$ exists in only e_1, e_2. Therefore v is a nest-point because $I^*(v, H) = \{e_1 \setminus \{v\}, e_2 \setminus \{v\}\}$ and $e_1 \setminus \{v\} \subset e_2 \setminus \{v\}$, so $I^*(v, H)$ is linearly ordered. Therefore we can only reduce again. Let $e_3 \subseteq e_4$ where it is possible for $e_4 = e_2$.

Step 3: Whether we can eliminate or reduce in the third step it is unclear if there would exists a 2-nest-set, or more importantly if the GYO algorithm succeeds. So we have to break our third step into the two possible cases and then look at the fourth step.

Step 4 (Case 1): Let us assume that we were able to eliminate in Step 3. If we can eliminate in Step 4 then either e_1 or e_3 is a nest-set. If not then we could have eliminated in Step 2 which we know implies that there exists a nest-point. If we reduce in Step 4 then one of the nested hyperedges must be a nest-set. If not then we would have a 3-cycle nested in a hyperedge with 3 vertices, meaning that we could have eliminated in Step 1, so there exists a nest-point.

Step 4 (Case 2): Let us assume that we were able to reduce in Step 3. If we can eliminate then we either could have eliminated in Step 1 or one of our nested hyperedges is a nest-set. If not then there would have been a vertex, v that exists in only one hyperedge in Step 2 which would mean that v is a nest-point. Note that we cannot reduce again since we assume H is connected and we do not repeat hyperedges.

Therefore if H is alpha acyclic there must exist a 2-NEO contradicting our assumption that there does not exist a 2-NEO.

□

In our above proposition we restricted the number of hyperedges to less than 5. This is because in hypergraphs with 6 or 7 hyperedges there exists a hypergraph that does not have a 2-NEO and is alpha acyclic. Let us examine these hypergraphs and see why this is the case.

Example 4.2.

Because we are looking at a hypergraph with 6 hyperedges we will have more

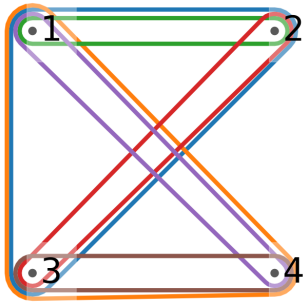


Figure 4.8: H where $|V(H)| = 4$ and $|E(H)| = 6$ such that H has no 2-NEO and is not alpha acyclic. This hypergraph was generated using code from Appendix A.

steps in our GYO algorithm. We can see that all of our hyperedges with 2 vertices are nested in a hyperedge with 3 vertices, so the GYO algorithm succeeds. Looking at Case 2 of our proof for Corollary 4.2 we can see that in Step 4, we would be able to reduce or eliminate, but that does not mean that our nested hyperedges are a nest-set. This is because all four of our nested hyperedges are connected, and reducing one does not result in a way to eliminate until all four have been reduced. This results in every combination of vertex sets, s, with at most 2 vertices to have an $I^*(s, H)$ with two singleton sets. So $I^*(s, H)$ will never be linearly ordered.

Example 4.3.

Observe that if we remove $\{1, 3\}$ then we have a hypergraph that is isomorphic to our hypergraph in Example 4.2. Since $\{1, 3\}$ is nested it does not affect alpha acyclicity and since it has a subhypergraph that does not have a 2-NEO adding an edge does not result in a hypergraph with a 2-NEO.

While all of our observations above are based on hypergraphs with 4

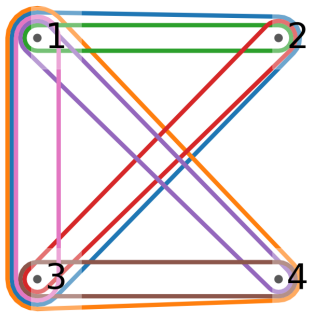


Figure 4.9: H where $|V(H)| = 4$ and $|E(H)| = 7$ such that H has no 2-NEO and is not alpha acyclic. This hypergraph was generated using code from Appendix A.

vertices we first evaluated hypergraphs with 6 vertices, building off of the example given in Lanzinger's paper, Figure 4.6. However, generating all hypergraph with 6 vertices proved to be a daunting task, especially when no longer looking at hypergraphs with full hyperedges. Despite this we were still able to gather information about these hypergraphs.

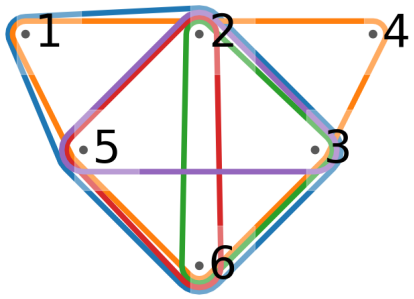


Figure 4.10: Hypergraph with 6 vertices a 2-nest-set and no 2-NEO.

The propositions mentioned earlier in this section were all iterations of previous conjectures that were proven to be false in general. We conjectured that for a hypergraph with 6 vertices if there exists a 2-nest-set then there exists a 2-NEO, Figure 4.10. However, this proved to be false once we were able to evaluate more hypergraphs.

The hereditary property of nest-set elimination orderings is what makes evaluation of hypergraphs with small order interesting. If any hypergraph has a subhypergraph that is not a 2-NEO , or k-NEO in more general terms, then that hypergraphs cannot be a 2-NEO or k-NEO respectively. These results can provide insight into a lower bound of the number of 2-NEOs for hypergraphs with a specific order and size.

Chapter 5

Databases and Graphs

There is a strong connection between hypergraphs and relational databases. This is because we can represent a database schema as a hypergraph, as seen in Figure 5.1. By doing so, our database schema can take on hypergraph properties such as acyclicity and nest-set width. In this chapter, we will first discuss the usefulness of acyclicity and joins, and then we will shift our focus to discuss nest-set width and querying.

Recall a database is an organized collection of structured data that is usually stored electronically and controlled by a database management system (DBMS). A relational database is a database that is structured to recognize relations between data, relation schemas, and we represent the structure of a relational database with a database schema. A database schema is an abstract design that represents the storage of data in our database and describes the organization and relationships between the relation schemas of a

database.

Because data schemas can be represented as hypergraphs we define a database schema to be acyclic in the same way we have defined alpha-acyclicity and beta-acyclicity. This leads us to the following definition from [25].

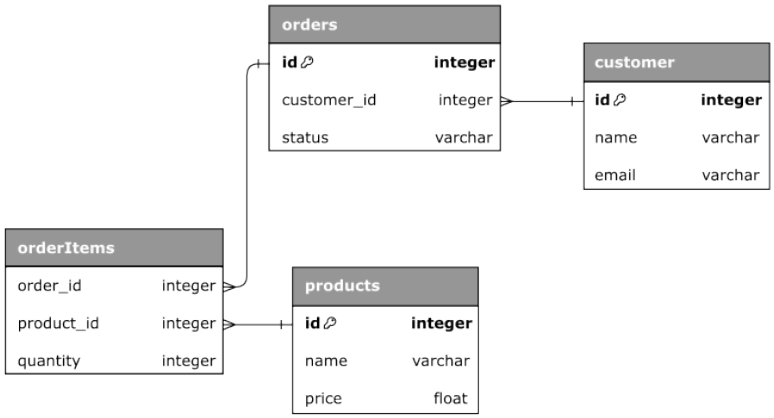
Definition 5.1. A database schema $\mathbf{R}$ is *acyclic* if and only if applying the following two operations repeatedly on the hypergraph H, induced by $\mathbf{R}$, results in a hypergraph having an empty set of hyperedges:

1. If an attribute A appears in exactly one hyperedge H then remove A from that hyperedge; A is called an *isolated* attribute

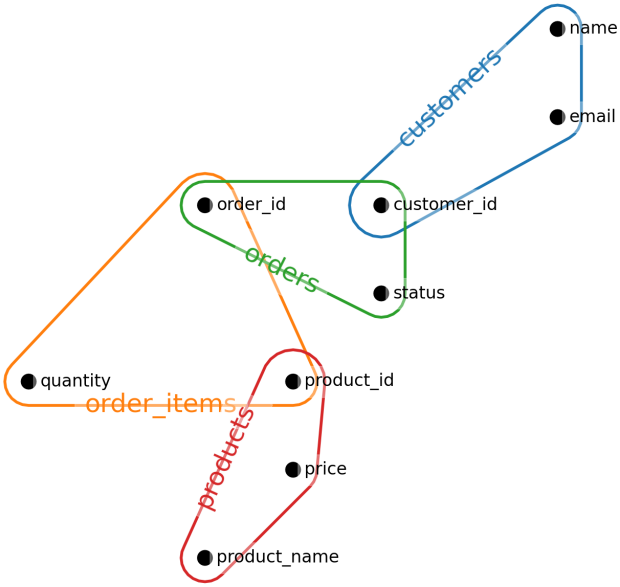
2. If S_i and S_j are distinct hyperedges in H such that $S_i \subseteq S_j$ then remove S_i from the hyperedge set of H; S_i is called a *redundant* relation schema.

A database schema $\mathbf{R}$ is *cyclic* if it is not acyclic.

This is the GYO algorithm that we discussed in Section 9. The GYO algorithm, and all other theorems, discussed with regards to alpha-acyclicity are an adaptation of the acyclicity of database schemas. First, we will discuss how acyclicity is useful in *joins*, an operation that combines two or more relations, and then we will describe how nest-set width is useful in queries, and requests for data from a database.



(a) Database schema



(b) Hypergraph of database schema

Figure 5.1: Our hypergraph in Figure 5.1b is based on Figure 5.1a where each hyperedge represents a relation schema in our data schema and the vertices are the entities of our database schema. Two hyperedges overlap if they share a vertex, i.e. entity.

5.1 Joins

The relationship between relational databases, chordal graphs, and their hypergraph generalizations has been studied and applied extensively. There has been a specific emphasis on database schemas and how databases can be designed to have a database schema that is alpha acyclic. Fagin defined alpha acyclic hypergraphs and beta acyclic hypergraphs and their connection with desirable properties of relational databases in 1983 [13], [14]. We will follow the structure, definitions, examples, and theorems from these two papers in this section.

Recall that every subhypergraph of a hypergraph that is beta acyclic is alpha acyclic, so every property $\mathcal{P}$ that we discuss applies to a database schema that is beta acyclic. This means a database schema is beta acyclic if and only if every subschema enjoys property $\mathcal{P}$. For the rest of this section acyclicity refers to alpha acyclicity.

5.1.1 Consistency

Let r and s be relations with attributes R and S respectively and $Q = R \cap S$, where Q is the set of attributes that r and s have in common. If $r[Q] = s[Q]$ then we say that r and s are *consistent*. In other words, the projections of r and s onto their common attributes are the same. We can see that Tables 5.1, 5.2, and 5.3 are consistent with their universal relation Table 5.4.

Let $\mathbf{r} = \{r_1, \ldots, r_n\}$ be an arbitrary database over $\mathbf{R} = \{R_1, \ldots, R_n\}$,

Table 5.1: Relation r_1

A	B	C
0	0	1
1	0	1
2	3	4

Table 5.2: Relation r_2

B	C	D
0	1	1
3	4	5

Table 5.3: Relation r_3

A	D
0	1
1	1
2	5

Table 5.4: "Universal" Relation

A	B	C	D
0	0	1	1
1	0	1	1
2	3	4	5

then **r** is *pairwise consistent* if each pair r_i and r_j is consistent. We call **r** *globally consistent* if there is a relation r over attributes $N = R_1 \cup \cdots \cup R_n$ such that $r_i = r[R_i]$ for each i. In other words, if **r** is globally consistent then there is a "universal relation" r such that each r_i is a projection over r.

Every consistent database is pairwise consistent, but the converse does not hold. However, if we have a database schema that is acyclic then the converse does hold. That is, if we have a database schema that is acyclic, then the database is consistent if and only if it is pairwise consistent. And every pair of relations is globally consistent if and only if the two relations are consistent. This leads us to our desired property that is equivalent to acyclicity which is,

Theorem 5.1. *A database is acyclic if and only if every pairwise consistent database over* **R** *is globally consistent.*

Understanding consistency, and more specifically global consistency is important because it predicts if a set of relations is joinable. This leads us

to our next topic, semijoins.

5.1.2 Semijoins

There are different strategies for obtaining the join of two relations. The semijoin is a specific type of join that allows for more efficient execution of joins. We denote the *semijoin* between relations r and s over attributes R and S respectively, as $r \ltimes s$, and define it as $(r \bowtie s)[R]$. We will show how the semijoin works by joining relation r_1 and r_2 from Table 5.1 and Table 5.2.

Step 1: Ship the BC projection from r_2 to r_1 (since BC is the only attribute that r_1 and r_2 have in common).

Step 2: Prune relation r_1 by removing the tuples whose projection A is not in the relation from Step 1. For example, (0,1) comes from the projection so we do not remove it. Observe that in this example we have no tuple that needs to be pruned.

Step 3: Ship the pruned r_1 relation to r_2 and take the join.

Semijoins are useful because they reduce the amount of communication by sending only the projected join column(s) to be joined with the second relation, however, this benefit only occurs when the database schema is acyclic. To show why cyclicity renders the semijoin strategy useless let us look at the database schema created by Tables 5.1 - 5.4. Observe that it is cyclic and

in Step 2 of our semijoin procedure, we were not able to prune any tuples. This inability to prune tuples is why there was no benefit from the semijoin strategy.

We define a *semijoin program* as a sequence of semijoin statements $r_i := r_i \ltimes r_j$. Given a database $\mathbf{r}$ we say that a semijoin program is a *full reducer* of all of the relations in the database if the semijoin program converts $\mathbf{r}$ into a globally consistent database. This leads to the following equivalent condition for α-acyclicity.

Theorem 5.2. *A schema is acyclic if and only if every database over* $\mathbf{R}$ *has a full reducer.*

We can summarize this result by saying that a database scheme is acyclic if and only if the semijoin strategy is useful.

5.1.3 Monotone Join Expressions

Let r_1, r_2, r_3, r_4 be relations (not associated with the above tables), over relation schemas R_1, R_2, R_3, R_4 respectively. If we were to take the join of all four relations the following may happen. Taking the join $r_1 \bowtie r_2$ may result in a thousand tuples and then taking that result with r_3 gets us, $r_1 \bowtie r_2 \bowtie r_3$ which may have a million tuples. Then taking the final join with r_4, $r_1 \bowtie r_2 \bowtie r_3 \bowtie r_4$ may result in only ten tuples, the desired result. This means that we have an intermediate step with thousand tuples and another with a million tuples. The way to remedy this is to use *monotone join*

expressions.

A *join expression* is a well-formed expression formed out of relation schemas, the symbol "⋈", and parenthesis, in which every join is binary. For example, Equation 5.1 is a join expression

$$((r_1 \bowtie r_4) \bowtie (r_2 \bowtie r_3)) \tag{5.1}$$

corresponding to joining r_2 and r_3 and then joining r_1 and r_4 before joining the results of the two. We call a join expression *monotone* with respect to a database $r_1, r_2, \ldots$ if every (binary) join that appears in the join expression is over consistent relations. So the join expression in 5.1 is monotone with respect to the database r_1, r_2, r_3, r_4 if:

1. r_1 and r_4 are consistent
2. r_2 and r_3 are consistent
3. $(r_1 \bowtie r_4)$ and $(r_2 \bowtie r_3)$ are consistent

If every join expression is monotone with respect to every pairwise consistent database then we have a *monotone join expression.* This leads us to the final condition that we will discuss in this section.

Theorem 5.3. *A database schema is acyclic if and only if there is a monotone join expression.*

Monotone join expressions ensure that no tuples get lost when taking the join. This means that it guarantees that no intermediate join has more tuples

than our final join, preventing the unpleasant behavior we described at the beginning of this subsection.

5.2 Queries

The usefulness of nest-set width is seen in the tractability of conjunctive queries with negation, which we will define later in this section, and the fixed-parameter tractability of SAT parameterized by nest-set-width. In this section we will provide a broad overview of the algorithm presented by Lanzigner [24]. For proofs and a more in-depth discussion of why the algorithm works readers are encouraged to read Lanzinger's paper.

Recall our discussion on queries in Chapter 2.3.2. Throughout this section we will highlight a commonly used class of queries called conjunctive queries also referred to as project-select-join queries. A *conjunctive query*, CQ, is a relational algebra query that is a finite composition of selections with only simple selection formulae, projections, and joins [25].

Example 5.1. Let us look at Figure 5.1a to construct some examples of conjunctive queries.

(a) Who created the order with order_id 100?

(b) What is the email of Jane Smith?

(c) What products are in the order with order_id 232?

(d) List all orders that Jane Smith has placed.

There are various equivalent versions of conjunctive queries. First, we will focus on the SPJR algebra, which stands for selection, projection, join, and renaming, as it follows the most closely with how we defined relational databases in Chapter 2.3. Then we will introduce the rule-based conjunctive query which has a clear relation to first-order logic. In both cases, we will follow the definition from Abiteboul's book *Foundations of Databases* [1].

Let us look at queries from Example 5.1 and how they are represented with an SPJR algebra.

Example 5.2. We have the following conjunctive queries based on Example 5.1:

(a) $\pi_{\text{customer\_id}}\left(\sigma_{\text{order\_id}=100}(\text{orders})\right)$

(b) $\pi_{\text{email}}\left(\sigma_{\text{name}=\text{`Jane Smith'}}(\text{customer})\right)$

(c) $\pi_{\text{product\_id}}\left(\sigma_{\text{order\_id}=232}(\text{orders})\right)$

(d) $\pi_{\text{order\_id}}\left(\sigma_{\text{name}=\text{`Jane Smith'}}(\text{orders}) \bowtie \text{customer}\right)$

We can express our queries as hypergraphs which we will denote as $H(q)$ where q is our query. The vertices of $H(q)$ are the attributes of q and $H(q)$ has an edge if and only if there exists a relation schema involved in the query. For the sake of simplicity, will assume that each relation occurs only once in a query.

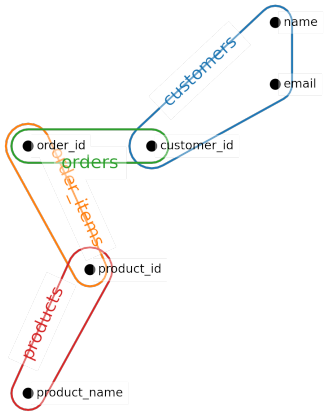


Figure 5.2: Hypergraph of our query in Example 5.3.

Example 5.3. Let us look at the conjunctive query "What is the email and name of people who have ordered cookies?"

$$\pi_{\text{email, name}}(((\sigma_{\text{name}=\text{`cookies'}}(\text{products}) \bowtie \text{order\_items}) \bowtie \text{orders}) \bowtie \text{customers})$$

If we have the query, "Did Jane Smith order cookies?" then we are looking for a simple yes or no answer. We do not need to return every single order_id with Jane Smith's customer_id or even just the order_id's with the product_id for cookies in it. However, we need this information to get an answer for the query. We will answer yes if there are orders that meet our condition, orders made by Jane Smith that have cookies in them, and no if we get no such orders. This query is a special kind of query called a boolean query, BQ. We define a BQ as a query that returns either "true", having a result, or "false", not having a result [9]. A boolean conjunctive query, BCQ, is a conjunctive query that returns either "true" or "false". Determining if there

exists an interpretation that satisfies a boolean formula is known as SAT, the satisfiability problem which we will discuss later in this section, and was the first problem proven to be NP-complete by Cook [10] and Levine [26].

Now we will discuss a logic-based view of conjunctive queries using first-order logic. We will specifically look at the rule based-conjunctive query because it is well suited for specifying queries from database schemas to database schemas.

Definition 5.2. Let $\mathbf{R}$ be a database schema. A *rule-based conjunctive query* over $\mathbf{R}$ is an expression of the form

$$ans(\vec{u}) \leftarrow R_1(\vec{u_1}), \ldots, R_n(\vec{u_n})$$

where $n \geq 0$, $R_1, \ldots, R_n$ are relation names in $\mathbf{R}$ and ans is a relation name not in $\mathbf{R}$. We say $\vec{u}, \vec{u_1}, \ldots, \vec{u_n}$ are free tuples (i.e. can use either variables or constants), such that if $\vec{v} = (v_1, \ldots, v_n)$, then $R(\vec{v}) = R(v_1, \ldots v_n)$. And each variable occurring in $\vec{u}$ must appear at least once in $\vec{u_1}, \ldots, \vec{u_n}$.

In first-order logic, we call $R(\vec{v})$ an atom, or literal which is positive or negative depending on whether we have an atom or negation of an atom. R is our predicate and $\vec{v}$ are our variables. Observe that we have been referring to $R(\vec{v})$ as a relation schema, so $\vec{v}$ are our attributes. For a query q we we will denote a set of variables, attributes, that occur in the literal, relation schema, of q as vars(q).

Example 5.4. We will simplify the attributes of the relation schemas from

Figure 5.1b as: orders(O, C, S), customer(C, N, E), order_items(O, P, Q), products(P, M). From Example 5.1 we have the following queries:

(a) $q_1(C) : ans \leftarrow \text{orders}(100, C, S)$

(b) $q_2(E) : ans \leftarrow \text{customers}(C, \text{'Jane Smith'}, E)$

(c) $q_3(P) : ans \leftarrow \text{order\_items}(232, P, Q)$

(d) $q_4(O) : ans \leftarrow \text{orders}(O, C, S), \text{customers}(C, \text{'Jane Smith'}, E)$

From Example 5.3

$$q_4(E, N) : ans \leftarrow \text{customers}(C, N, E), \text{orders}(O, C, S), \\ \text{order\_items}(O, P, Q), \text{products}(P, \text{'cookies'}, M)$$

From Example 5.1

$$q_5(O) : ans \leftarrow \text{customers}(C, \text{'Jane Smith'}, E), \text{orders}(O, C, S)$$

CQs have been extensively studied and have been shown to be tractable [29], solvable in practice, when the tree-likeness of alpha acyclic hypergraphs are generalized by hypertree width and fractional hypertree width [23]. However, few results extend to conjunctive queries with negation, CQ$^\neg$, even though the use of negation in queries makes the hypergraph of the query have a simpler structure.

Example 5.5. The following are negative conjunctive queries:

(a) What orders do not have cookies in them?

(b) What orders are not pending?

Writing queries as an SPJR algebra we have:

(a) $\pi_{\text{order\_id}}(\sigma_{\neg(\text{name}=\text{`cookies'}}(\text{products}) \bowtie \text{order\_items})$

(b) $\pi_{\text{order\_id}}(\sigma_{\neg(\text{status}=\text{`pending'})}(\text{orders}))$

Writing the queries as a rule-based conjunctive query we have:

(a) $q_1 : ans(O) \leftarrow \neg\text{products}(P, \text{`cookies'}), M), \text{order\_items}(O, P, Q)$

(b) $q_2 : ans(O) \leftarrow \neg\text{orders}(O, C, \text{`pending'})$

Now that we know how to represent conjunctive queries we can discuss how boolean conjunctive queries with negation, BCQ$^\neg$'s, are tractable when bounded by nest-set-width. An algorithm established by Brault-Baron [3] discusses how beta-acyclicity makes CQ$^\neg$'s easy, however, the overall approach of this algorithm does not generalize to nest-set width. This is when it becomes important to create a distinction between 1-nest-sets and nest-sets of larger size resulting in a new algorithm established by Lanzinger whose paper we will follow for the rest of the section [24].

Before we continue we must state some simplifying assumptions. We assume queries in nest-set width instances are safe, no attribute occurs only in a negative literal. An unsafe query can always be made safe and the

subqueries created in the algorithm may not always be safe. Assume the size of the domain of the database, Dom is a power of 2, $|\text{Dom}| = 2^d, d \in \mathbb{Z}$.

The algorithm for BCQ$^\neg$ revolves around the successive elimination of variables from the query. We determine which variables to eliminate based on the nest-set elimination ordering, eliminating all variables of a nest-set at once. The elimination of a nest-set s can be done in three steps:

1. Eliminate all occurrences of variables from s in positive literals.

2. Extend the negative literals to s in such a way that they form a beta-acyclic subquery.

3. Eliminate the variables of s from the beta-acyclic subquery.

Before we provide an example of how these steps work there are some definitions and lemmas that we need to state.

Definition 5.3. Let $R(v_1, \ldots, v_n)$ be a literal and s a set of variables. Let $s' = s \setminus \{v_1, \ldots, v_n\}$ be the variables in s that are not used in the literal. We call the new literal $R(v_1, \ldots, v_n, s')$ the *s-extension* of R.

Definition 5.4. We call a function $a : \text{vars}(q) \to \text{Dom}$ an *assignment* for q. For a set of variables X we write $a[X]$ for the assignment with domain restricted to X.

Lemma 5.1. *Let $\vec{x}, \vec{s}$ be lists of variables and $R(\vec{x}, \vec{s})$ be a positive literal on the relation* **R**. *Define new literals $P(\vec{x})$ where* **P** *is the relation created*

by the projection $\pi_{\vec{x}}(\mathbf{R})$ *and* $\neg C(\vec{x}, \vec{s})$ *where* $\mathbf{C} = \mathbf{P_s} \setminus \mathbf{R}$*, where* P_s *is the extension of* P*. Then an assignment satisfies* $R(\vec{x}, \vec{s})$ *if and only if it satisfies* $P(\vec{x}) \wedge \neg C(\vec{x}, \vec{s})$*.*

Lemma 5.2. *Let* q *be the query consisting of the literals*

$$\{\neg R_1(\vec{x}_1, y), \neg R_2(\vec{x}_2, y), \ldots, \neg R_n(\vec{x}_n, y)\}$$

on a database D *with domain* $|Dom| = 2^k$ *and let* $\{y\}$ *be a nest-set of* q*. There exists a query* q' *of the form* $\neg R_1(\vec{x}_1), \neg R_2(\vec{x}_2), \ldots, \neg R_n(\vec{x}_2)$ *and a database* D' *with the following properties:*

1. *If* $q \in q(D)$*, then* $a[va(q')] \in q'(D)$*.*
2. *If* $a' \in q'(D')$ *then there exists* $c \in Dom$ *such that* $a' \cup \{y \mapsto c\} \in q(D)$*.*
3. q' *and* D' *can be computed in* $O(||D|| \log^2 |Dom|)$ *time given* q *and* D *as input.*
4. *For every* $i \in [n]$ *we have* $|\mathbf{S}_i| \leq |\mathbf{R}_i|$*. Where* $\mathbf{S}$ *are the relations in* D' *and* $\mathbf{R}$ *are the relations in* D*.*

Now we have all the tools to show an example using the algorithm presented by Lanzinger.

Example 5.6 ([24])**.** Let q be a query with nest-set $s = \{a, b, c\}$. The query has literals $P_1(a, b, c), P_2(b, d), \neg N_1(a, d, e, f, g), \neg N_2(c, d, e)$ incident to s. See

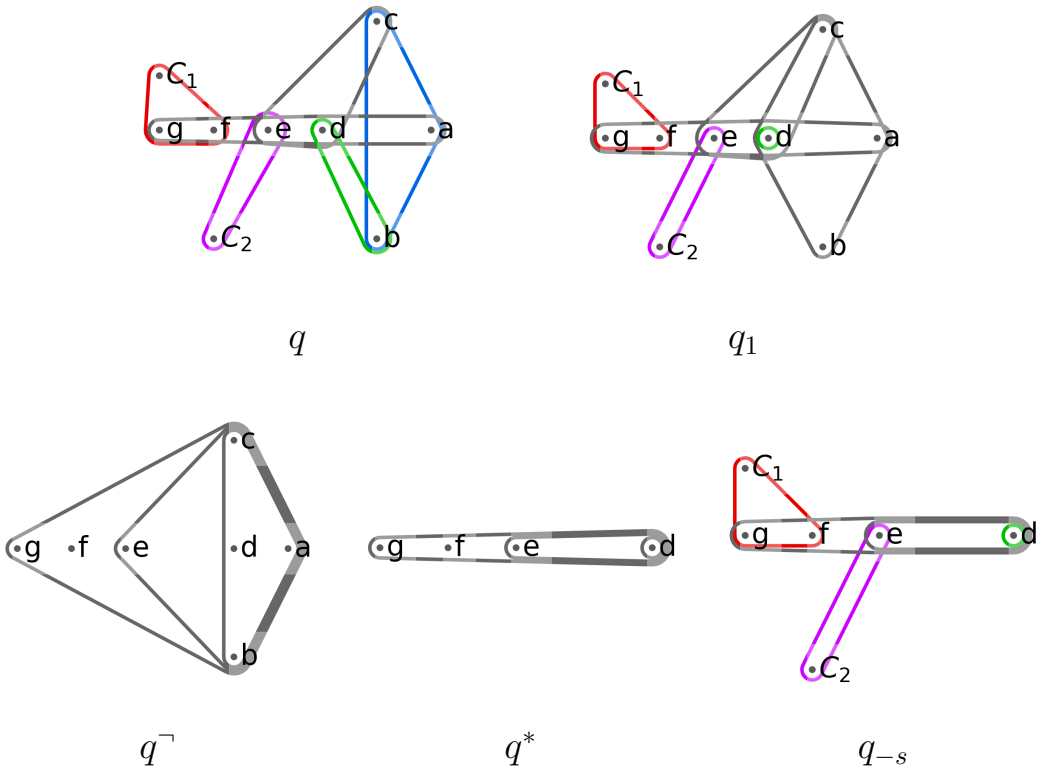


Figure 5.3: Example of an s-elimination on a hypergraph. Negative literals are represented by gray hyperedges.

Figure 5.3 to see the process on the hypergraph level. We will use C_1 and C_2 to denote the rest of the query.

Elimination of s from positive literals: Let q_J be the subquery containing all the incident literals. To compute what q_J outputs we can join them together using an anti-join. Let J be a new relation that consists of the solutions of q_J, leaving us with $J(a, b, c, d)$. Let P be the projection of schema$(J) \setminus s$ on J, so that $P = \pi_{\text{vars}(J)\setminus s}(J)$ resulting in a new literal $P(d)$. Taking the s extension of P we get $P_s = \{a, b, c, d\}$ and

define another new relation $C = P_s \setminus J$ and negate it so that we have $\neg C(a, b, c, d)$. Now we can define a new query $q_1 = (q \setminus q_J) \cup \{P, \neg C\}$. Our new query $q_1 = q$ has all elements of s in a negative literal.

Elimination of s from negative literals: Let us now look at the negative literals that are incident to s, $\neg N_1, \neg N_2, \neg C$. We need to find their s-extensions to expand them over s; $\neg N_1'(a, d, e, f, g, b, c)$, $\neg N_2'(c, d, e, a, b)$, $\neg C'(a, b, c, d)$ are our s-extensions. Let these s-extensions make up a new subquery $q^\neg$. The hypergraph structure of $q^\neg$ is a query where every single element of s is now a 1-nest-set, and can now be eliminated using 5.2 to obtain q^*.

$$q^* = \{\neg C^*(d), \neg N_1^*(d, e, f, g), \neg N_2^*(d, e)\}$$

Replacing P with $P = C'$ and $q^\neg$ with q^* we get a new query,

$$q_{-s} = (q_1 \setminus q^\neg) \cup \{N_1', \ldots, N_2'\}$$

We are now left with a query, q_{-s} without variables from s, the s-elimination of q.

Lazinger shows that an s-elimination has a solution if and only if the original query is a solution and that it can be computed in polynomial time when s is bounded. It then follows that the s-elimination is always smaller than the original query. It is important to note that our simplifying assumptions

are preserved when we construct an s-elimination, so the domain is never modified and q is safe. Additionally, we can perform this elimination repeatedly along with a k-NEO since $H(q') = H(q) \setminus s$, allowing us to determine whether $q(D) \neq \emptyset$. These properties get us our main result.

Theorem 5.4. *Let Q be a class of BCQ$^\neg$ instances with bounded nest-set width. Then BCQ$^\neg$[Q] is decidable in polynomial time.*

The satisfiability problem (SAT) is the problem of determining whether there exists a truth assignment that satisfies a set of clauses, where a clause is a set of literals, such that at least one member of each clause has the value true [18]. If the answer is 'yes' then we say that I has been satisfied. Lanzinger shows that there exists a many-one reduction r from SAT to BCQ$^\neg$ such that for every formula F the hypergraph $H(F)$ is the same as the hypergraph of the query in $r(F)$. This fact along with Theorem 5.4 shows that SAT is tractable when restricted to classes of formulas with constantly bounded nsw. However, we can expand on this and get fixed-parameter tractability.

Theorem 5.5. *SAT for propositional conjunctive normal form formulas is fixed-parameter tractable when parameterized by the nest-set width of the formula.*

The ability of nest-set width to maintain these properties of beta-acyclicity is what makes it such an interesting and important generalization. There is

still a lot more to explore with nest-set width including the relationship between it and another beta-acyclicity generalization point-width [8]. However, the contributions of Lanzinger's paper give us a way to generalize tractability, improving upon previous generalizations.

Chapter 6

Conclusion

The relationship between graph theory and database theory, particularly relational databases, has been a subject of study since the 1980s, with a specific focus on alpha acyclicity. Acyclicity in database schemas provides properties that make many NP problems more manageable, such as conjunctive queries with negation and boolean conjunctive queries with negation. The aim has been to extend these properties to more hypergraphs, leading to generalizations of alpha acyclicity and beta acyclicity.

Although the focus has been mainly on alpha acyclicity, there have been attempts to generalize beta acyclicity. However attempts of generalization do not retain the same tractability that we get from beta acyclic hypergraphs. This is why the introduction of nest-set width, which generalizes tractability results for beta acyclicity, is so interesting.

In this thesis, we explored the acyclicity of hypergraphs with a particular

focus on alpha acyclicity and beta acyclicity. We demonstrated the connection between elimination orderings of acyclic hypergraphs and elimination orderings for chordal and strongly chordal graphs. This motivated us to define a generalization of beta acyclicity called nest-set width, where again we saw the usefulness of elimination orderings and their generalizations.

Bounding queries by nest-set width gets us tractability of boolean conjunctive queries with negation and fixed-parameter tractability of SAT. These results were the motivation for exploring nest-set elimination orderings in more depth. However, we first needed to construct a dataset of hypergraphs categorized by the number of vertices, number of hyperedges, and size of each hyperedge. From this dataset, we were able to determine small hypergraphs that always have a 2-NEO. We also found a correlation between alpha acyclicity and small hypergraphs without a 2-NEO.

Since nest-set elimination orderings are hereditary, evaluating them on smaller hypergraphs raises the interesting question of whether there is a potential lower bound on the number of 2-NEO's that exist. For all hypergraphs of order greater than 4, there will have to exist hypergraphs with subhypergraphs that do not have a 2-NEO. However, even without knowing a lower bound, understanding these hypergraphs helps in the construction of acyclic databases since we know what to avoid. Additionally, there is a question of whether line graphs of a hypergraph with a k-NEO are also chordal since the chordality of a line graph because this is true for beta acyclic hypergraphs and for the dual of alpha acyclic hypergraphs. Although this may not be

true for every k-NEO, there is potential for this to be true under certain restrictions, such as the number of vertices or hyperedges.

Additionally, the results obtained for nest-set width raise the question of whether or not it can be applied more broadly. Other problems have desirable complexities when the hypergraph structure is beta acyclic. One such example is #SAT, which is the problem of counting the number of truth-assignments that satisfy a set of clauses. While SAT asks whether a solution exists, #SAT asks how many solutions there are. It has been shown by Brault-Baron et al. [5] that #SAT on beta acyclic conjunctive normal forms can be solved in polynomial time. The algorithm to show this depends on nest-point elimination orderings, so nest-set elimination orderings are a natural candidate for a generalization of this result.

Appendix A

Generate Random Hypergraph

Many algorithms allow one to generate a random hypergraph, however, the following Python code was created based on direct sampling [15], where each hyperedge is constructed randomly based on the number of vertices and size of each hyperedge. Since we are working with hypergraphs with no repeated hyperedges we check to make sure that hypergraphs with repeated hyperedges are removed and duplicate hypergraphs are removed.

```
# --------
# Generate random hypergraphs
# --------

import random

def generate_random_hypergraph(numVertices, num_hyperedges,
```

```
    edge_size):
    if numVertices <= 0 or num_hyperedges <= 0 or len(edge_size) !=
        num_hyperedges:
        raise ValueError("Inputs must be positive integers")

    vertices = set(range(1, numVertices + 1))
    hyperedges = []

    for size in edge_size:
        verticesInEdge = random.sample(vertices, size)
        hyperedges.append(verticesInEdge)

    return hyperedges

# Remove any duplicate hypergraphs
def remove_outer_duplicates(list_of_lists_of_lists):
    seen = set()
    uniqueHypergraphs = []

    for element in list_of_lists_of_lists:
        # Convert inner lists to sets to disregard the order of
            elements
        element_sets = {frozenset(sublist) for sublist in element}
```

```
        # Check if the set representation of the element is already
            seen
        if tuple(element_sets) not in seen:
            uniqueHypergraphs.append(element)
            seen.add(tuple(element_sets))

    return remove_duplicate_edge(uniqueHypergraphs)

# Remove hypergraphs where with a repeated hyperedge
def remove_duplicate_edge(listHypergraphs):
    unique = []

    for sublist in listHypergraphs:
        # Convert each sublist to a set to remove duplicate elements
        unique_sublist = [list(set(inner_list)) for inner_list in
            sublist]

        # Check if there are any duplicates in the sublist
        if len(unique_sublist) == len(set(tuple(sub) for sub in
            unique_sublist)):
            # Add the unique sublist to the result
            unique.append(sublist)

    return unique
```

Appendix B

GYO Algorithm

The most common definition of an alpha-acyclic hypergraph is, a hypergraph is alpha-acyclic if and only if it can be fully reduced by the GYO algorithm. The following Python code is an interpretation of the GYO algorithm, following the steps highlighted in [19].

```
# --------
# GYO algorithm (Graham's algorithm) is an algorithm that finds if
    a hypergraph is alpha-acyclic or not
# --------

import copy

def gyo(hypergraph):
    # Run elimination and then reduction with hypergraph from
```

```
        elimination function
    ORIGINAL = copy.deepcopy(hypergraph)

    hyperedges = []
    for edges, vertices in hypergraph.items():
        hyperedges.append(vertices)

    reduction(hypergraph, hyperedges)

    # Keep repeating until you get the empty hypergraph or the
        hypergraph remains unchanged
    if bool(hypergraph) == False:
        return True
    elif ORIGINAL == hypergraph:
        return False
    else:
        return gyo(hypergraph)

def elimination(hypergraph):
    # Elimination of vertices that exist in one hyperedge
    count = 0

    hyperedges = []
    for edges, vertices in hypergraph.items():
```

```
            hyperedges.append(vertices)

        list_vertices = []
        for i in range(0, len(hyperedges)):
            list_vertices = list(set(hyperedges[i] + list_vertices))



        for i in range(0, len(list_vertices)):
            # Elimination of vertices that appear in one edge
            vertex = list_vertices[i]
            for edges, vertices in hypergraph.items():
                if vertex in vertices:
                    count += 1

            if count == 1:
                for edges, vertices in hypergraph.items():
                    if vertex in vertices:
                        vertices.remove(vertex)

            count = 0

        return hypergraph

    def delete_hyperedge(hypergraph, hyperedge):
```

```
    edges = []
    for edge, vertices in dict(hypergraph).items():
        if vertices == hyperedge:
            edges.append(edge)

    del hypergraph[edges[0]]

    return hypergraph

def reduction(hypergraph, hyperedges):
    # Reduce hypergraph by removing nested hyperedges
    nested = []
    for i in range(len(hyperedges)):
        for j in range(len(hyperedges)):
            # Check if hyperedges are nested
            if i!= j and set(hyperedges[i]) <= set(hyperedges[j]):
                nested.append(hyperedges[i])
            elif not hyperedges[i]:
                delete_hyperedge(hypergraph, hyperedges[i])

    if len(nested) != 0:
        delete_hyperedge(hypergraph, nested[0])

    return elimination(hypergraph)
```

Appendix C

Beta-Acyclicity

To find beta-acyclic hypergraphs we translated the first characterization, beta-leaf elimination orders, from [4] into Python.

```
# --------
# Algorithm to determine if a graph is beta-acyclic
# 1. Remove vertices that exist in one hyperedge
# 2. Find nest points and remove them
# 3. Remove empty edges
# 4. Repeat until hypergraph is empty or is unchanged
# --------

import copy

def check_beta_acyclic(hypergraph):
    ORIGINAL = copy.deepcopy(hypergraph)
```

```
    find_nest_point(hypergraph)

    # Keep repeating until you get the empty hypergraph or
        hypergraph remains unchanged
    if bool(hypergraph) == False:
        return True
    elif ORIGINAL == hypergraph:
        return False
    else:
        return check_beta_acyclic(hypergraph)

def find_nest_point(hypergraph):
    # Find the nest points
    hyperedges = []

    for edges, vertices in hypergraph.items():
        hyperedges.append(vertices)

    nodes = []
    for i in range(len(hyperedges)):
        nodes = list(set(hyperedges[i] + nodes))

    for i in range(len(nodes)):
```

```
        edge_length = {}
        for j in range(len(hyperedges)):
            if nodes[i] in hyperedges[j]:
                edge_length[j] = len(hyperedges[j])

        sort = dict(sorted(edge_length.items(), key=lambda item:
            item[1]))

        if check_nested(sort, hyperedges) == True:
            # If node is a nest point then remove it from H
            for edges, vertices in hypergraph.items():
                # Remove node (vertex)
                if nodes[i] in vertices:
                    vertices.remove(nodes[i])
            break

    return delete_empty_edge(hypergraph)

def check_nested(dict, list_edges):
    # Check if a list of lists is nested
    count = 0

    dictKeys = list(dict.keys())
    edgeIndex = []
```

```
    for i in range(0, len(dictKeys)):
        edgeIndex.append(dictKeys[i])

    for i in range(len(dictKeys)-1):
        # Check if one edge is in another
        if set(list_edges[edgeIndex[i]]) <=
            set(list_edges[edgeIndex[i+1]]) or
            set(list_edges[edgeIndex[i]]) >=
            set(list_edges[edgeIndex[i+1]]):
            count += 1
    if count == (len(edgeIndex) - 1):
        return True
    else:
        return False

def delete_empty_edge(hypergraph):
    # Check if edge is empty
    for edges, vertices in dict(hypergraph).items():
        if len(vertices) == 0:
            del hypergraph[edges]

    return hypergraph
```

Appendix D

Find NEO

To find if a hypergraph has a nest-set elimination ordering, we followed the definition of a nest-set-elimination ordering as highlighted in [24]. Note that there is an optimized algorithm highlighted in the paper, however, that is not reflected in the proceeding Python code.

```
# --------
# Check if a hypergraph has a nest-set elimination ordering of a
    specified size (So it can also be used to check for beta
    acyclicity)
# --------

import itertools, copy

from nest_point_elim import delete_empty_edge
```

```
def neo(hypergraph, size):
    ORIGINAL = copy.deepcopy(hypergraph)

    get_set(hypergraph, size)

    # Keep repeating until you get the empty hypergraph or the
       hypergraph remains unchanged
    if bool(hypergraph) == False:
        return True
    elif ORIGINAL == hypergraph:
        return False
    else:
        return neo(hypergraph, size)

def get_set(hypergraph, size):
    # Find all possible nest sets
    hyperedges = []

    for edges, vertices in hypergraph.items():
        hyperedges.append(vertices)

    nodes = []
    for i in range(len(hyperedges)):
```

```
            nodes = list(set(hyperedges[i] + nodes))

        list_sets = []
        for i in range(size, 0, -1):
            sub_list_sets = [list(x) for x in
                itertools.combinations(nodes, i)]
            list_sets.extend(sub_list_sets)

        if not list_sets:
            return delete_empty_edge(hypergraph)
        else:
            return remove_vertices(hypergraph, is_nest_set(list_sets,
                hyperedges))

    def is_nest_set(list_sets, hyperedges):
        # Determine if a set is a nest set
        nestSets = []

        for i in range(len(list_sets)):
            set = list_sets[i]
            I = [] # Set of incident edges
            for edge in hyperedges:
                if any(i in edge for i in set) and edge not in I:
                    I.append(edge)
```

```
        for vertex in set:
            IReduced = [[node for node in edge if node != vertex]
                for edge in I]
            I = IReduced
        if linearly_ordered(I):
            nestSets.append(set)

    if not nestSets:
        return nestSets
    else:
        return nestSets[0]

def linearly_ordered(list_sets):
    # Check a list of sets to see if it is linearly ordered
    list_sets.sort(key = len)
    count = 0
    for i in range(len(list_sets)-1):
        if set(list_sets[i]) <= set(list_sets[i+1]) or
            set(list_sets[i]) >= set(list_sets[i+1]):
            count += 1

    if count == (len(list_sets)-1):
        return True
```

```
    else:
        return False


def remove_vertices(hypergraph, set_vertices):
   # Remove vertices that are in a certain nest-set
    for vertex in set_vertices:
        for edges, vertices in hypergraph.items():
            if vertex in vertices:
                vertices.remove(vertex)


    return hypergraph
```

Appendix E

Hypergraph Database and Queries

To evaluate hypergraphs and make our observations we needed to utilize the code in the previous appendices to generate and evaluate hypergraphs. A database of over 28000 hypergraphs was generated, mainly focused on hypergraphs with a small size, hypergraphs with 6 or fewer vertices. While not every single possible hypergraph is generated for hypergraphs with 6 or fewer vertices, the database does contain all hypergraphs up to isomorphism for hypergraphs with 3 or 4 vertices.

One can explore the Google Colab: Hypergraph Analysis, which contains queries used with the database and provides access to it. And can look to the GitHub page: Algorithmic Graph Theory, to access the Python files of the proceeding appendices as well as additional code.

Bibliography

[1] S. (Serge) Abiteboul, Richard Hull, and Victor Vianu. **Foundations of Databases**. Addison-Wesley, 1995. 714 pp. ISBN: 978-0-201-53771-0 (cit. on p. 69).

[2] Catriel Beeri et al. "On the Desirability of Acyclic Database Schemes". In: *Journal of the ACM* 30.3 (July 1983), pp. 479–513. ISSN: 0004-5411, 1557-735X. DOI: 10.1145/2402.322389 (cit. on pp. 2, 28).

[3] Johann Brault-Baron. "A Negative Conjunctive Query Is Easy If and Only If It Is Beta-Acyclic". In: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2012, 15 pages. ISBN: 978-3-939897-42-2. MR: 3059545 (cit. on pp. 2, 73).

[4] Johann Brault-Baron. "Hypergraph Acyclicity Revisited". In: *ACM Computing Surveys* 49.3 (2016), p. 26. ISSN: 0360-0300, 1557-7341. DOI: 10.1145/2983573 (cit. on pp. 6, 25, 28, 30, 34, 90).

[5] Johann Brault-Baron, Florent Capelli, and Stefan Mengel. "Understanding Model Counting for β-Acyclic CNF-formulas". In: *32nd In-*

ternational Symposium on Theoretical Aspects of Computer Science (STACS 2015). Schloss Dagstuhl, Feb. 1, 2015, pp. 143–156. ISBN: 978-3-939897-78-1 (cit. on p. 82).

[6] Alain Bretto. **Hypergraph Theory: An Introduction**. Mathematical Engineering. Springer, Cham, 2013. xiv+119. ISBN: 978-3-319-00079-4. DOI: 10.1007/978-3-319-00080-0. MR: 3077516 (cit. on pp. 10, 37).

[7] Andries E. Brouwer and Antoon W. J. Kolen. **A Super-Balanced Hypergraph Has a Nest Point**. Mathematisch Centrum, Amsterdam, 1980. i+7. MR: 600103 (cit. on p. 36).

[8] Clément Carbonnel, Miguel Romero, and Stanislav Živný. "Point-Width and Max-CSPs". In: *2019 34th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*. IEEE, [Piscataway], NJ, 2019, [13 pp.] ISBN: 978-1-72813-608-0. MR: 4142430 (cit. on pp. 39, 79).

[9] Ashok K. Chandra and Philip M. Merlin. "Optimal Implementation of Conjunctive Queries in Relational Data Bases". In: *Conference Record of the Ninth Annual ACM Symposium on Theory of Computing (Boulder, Colo., 1977)*. Association for Computing Machinery, New York, 1977, pp. 77–90. MR: 489103 (cit. on p. 70).

[10] Stephen A. Cook. "The Complexity of Theorem-Proving Procedures". In: *Proceedings of the Third Annual ACM Symposium on Theory of Computing*. STOC '71. New York, NY, USA: Association for Com-

puting Machinery, May 3, 1971, pp. 151–158. ISBN: 978-1-4503-7464-4. DOI: 10.1145/800157.805047 (cit. on pp. 16, 71).

[11] Alessandro D'Atri and Marina Moscarini. "On Hypergraph Acyclicity and Graph Chordality". In: *Information Processing Letters* 29.5 (1988), pp. 271–274. ISSN: 0020-0190,1872-6119. DOI: 10.1016/0020-0190(88)90121-4. MR: 981077 (cit. on p. 37).

[12] Reinhard Diestel. **Graph Theory**. Fifth. Vol. 173. Graduate Texts in Mathematics. Springer, Berlin, 2017. xviii+428. ISBN: 978-3-662-53621-6. DOI: 10.1007/978-3-662-53622-3. MR: 3644391 (cit. on p. 4).

[13] Ronald Fagin. "Acyclic Database Schemes (of Various Degrees): A Painless Introduction". In: *CAAP '83 (L'Aquila, 1983)*. Vol. 159. Lecture Notes in Comput. Sci. Springer, Berlin , 1983, pp. 65–89. ISBN: 978-3-540-12727-7. DOI: 10.1007/3-540-12727-5_3. MR: 744202 (cit. on pp. 12, 63).

[14] Ronald Fagin. "Degrees of Acyclicity for Hypergraphs and Relational Database Schemes". In: *Journal of the Association for Computing Machinery* 30.3 (1983), pp. 514–550. ISSN: 0004-5411,1557-735X. DOI: 10.1145/2402.322390. MR: 709831 (cit. on p. 63).

[15] Bailey K. Fosdick et al. "Configuring Random Graph Models with Fixed Degree Sequences". In: *SIAM Review* 60.2 (2018), pp. 315–355. ISSN: 0036-1445,1095-7200. DOI: 10.1137/16M1087175. MR: 3797721 (cit. on p. 83).

[16] D. R. Fulkerson and O. A. Gross. “Incidence Matrices and Interval Graphs”. In: *Pacific Journal of Mathematics* 15 (1965), pp. 835–855. ISSN: 0030-8730,1945-5844. MR: 186421 (cit. on p. 7).

[17] Alan Gibbons. **Algorithmic Graph Theory**. Cambridge University Press, Cambridge, 1985. xii+259. ISBN: 0-521-24659-8. MR: 806954 (cit. on pp. 13, 15).

[18] Martin Charles Golumbic. **Algorithmic Graph Theory and Perfect Graphs**. Computer Science and Applied Mathematics. Academic Press [Harcourt Brace Jovanovich, Publishers], New York-London-Toronto, 1980. xx+284. ISBN: 978-0-12-289260-8. MR: 562306 (cit. on p. 78).

[19] Martin Charles Golumbic, Robin J. Wilson, and Lowell W. Beineke, eds. **Topics in Algorithmic Graph Theory**. Vol. 178. Encyclopedia of Mathematics and Its Applications. Cambridge University Press, Cambridge, 2021. xvi+349. ISBN: 978-1-108-49260-7. DOI: 10.1017/9781108592376. MR: 4273614 (cit. on pp. 6–9, 28, 37, 86).

[20] Nathan Goodman and Oded Shmueli. “Syntactic Characterization of Tree Database Schemas”. In: *Journal of the Association for Computing Machinery* 30.4 (1983), pp. 767–786. ISSN: 0004-5411,1557-735X. DOI: 10.1145/2157.322405. MR: 819130 (cit. on p. 28).

[21] Georg Gottlob, Nicola Leone, and Francesco Scarcello. “Hypertree Decompositions and Tractable Queries”. In: *Proceedings of the Eighteenth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database*

Systems. PODS '99. New York, NY, USA: Association for Computing Machinery, May 1, 1999, pp. 21–32. ISBN: 978-1-58113-062-1. DOI: 10.1145/303976.303979 (cit. on p. 39).

[22] Georg Gottlob and Reinhard Pichler. "Hypergraphs in Model Checking: Acyclicity and Hypertree-Width versus Clique-Width". In: *Automata, Languages and Programming*. Vol. 2076. Lecture Notes in Comput. Sci. Springer, Berlin, 2001, pp. 708–719. ISBN: 978-3-540-42287-7. DOI: 10.1007/3-540-48224-5_58. MR: 2066545 (cit. on p. 39).

[23] Martin Grohe and Dániel Marx. "Constraint Solving via Fractional Edge Covers". In: *Proceedings of the Seventeenth Annual ACM-SIAM Symposium on Discrete Algorithms*. ACM, New York, 2006, pp. 289–298. ISBN: 978-0-89871-605-4. DOI: 10.1145/1109557.1109590. MR: 2368823 (cit. on p. 72).

[24] Matthias Lanzinger. "Tractability beyond β-Acyclicity for Conjunctive Queries with Negation and SAT". In: *Theoretical Computer Science* 942 (2023), pp. 276–296. ISSN: 0304-3975,1879-2294. DOI: 10.1016/j.tcs.2022.12.002. MR: 4524096 (cit. on pp. 2, 25, 33, 39, 40, 44, 47, 51, 68, 73, 75, 94).

[25] Mark Levene and George Loizou. **A Guided Tour of Relational Databases and Beyond**. London: Springer, 1999. ISBN: 978-1-85233-008-8 (cit. on pp. 16, 61, 68).

[26] Leonid Anatolevich Levin. “Universal Sequential Search Problems”. In: *Problemy peredachi informatsii* 9.3 (1973), pp. 115–116 (cit. on p. 71).

[27] Eugene M. Luks. “Hypergraph Isomorphism and Structural Equivalence of Boolean Functions”. In: *Annual ACM Symposium on Theory of Computing (Atlanta, GA, 1999)*. ACM, New York, 1999, pp. 652–658. ISBN: 978-1-58113-067-6. DOI: 10.1145/301250.301427. MR: 1798088 (cit. on p. 48).

[28] M.H. Graham. “On the Universal Relation”. In: Technical Report (1979) (cit. on p. 27).

[29] Dániel Marx. “Tractable Hypergraph Properties for Constraint Satisfaction and Conjunctive Queries”. In: *Journal of the ACM* 60.6 (2013), Art. 42, 51. ISSN: 0004-5411,1557-735X. DOI: 10.1145/2535926. MR: 3144912 (cit. on p. 72).

[30] Asish Mukhopadhyay and Md. Zamilur Rahman. “Algorithms for Generating Strongly Chordal Graphs”. In: *Transactions on Computational Science XXXVIII*. Vol. 12620. Lecture Notes in Comput. Sci. Springer, Berlin, 2021, pp. 54–75. ISBN: 978-3-662-63170-6. DOI: 10.1007/978-3-662-63170-6_4. MR: 4248879 (cit. on p. 7).

[31] Mihalis Yannakakis. “Algorithms for Acyclic Database Schemes”. In: *Proceedings of the Seventh International Conference on Very Large Data Bases - Volume 7*. VLDB ’81. Cannes, France: VLDB Endowment, Sept. 9, 1981, pp. 82–94 (cit. on p. 2).

[32] C.T. Yu and M.Z. Ozsoyoglu. “An Algorithm for Tree-Query Membership of a Distributed Query”. In: *COMPSAC 79. Proceedings. Computer Software and The IEEE Computer Society's Third International Applications Conference, 1979.* COMPSAC 79. Proceedings. Computer Software and The IEEE Computer Society's Third International Applications Conference, 1979. Nov. 1979, pp. 306–312. DOI: 10.1109/CMPSAC.1979.762509 (cit. on p. 27).