Low-Depth Circuit Size Bounds Using Combinatorial Methods

Levente Bodnár

St Anne's College

University of Oxford

A thesis submitted for the degree of

Doctor of Philosophy

Trinity 2025

Abstract

This thesis focuses on constant-depth circuit size lower bounds for the clique, threshold and parity functions. The thesis is centred around the combinatorial tools and questions relevant for attacking these circuit size lower bound problems.

First, the analysis will focus on the maximum one-sided correlation achievable between polynomial-sized, constant-depth circuits and the clique function. Expressing the AND of edges in a given clique provides a $\binom{n}{k}^{-1}$ one-sided correlation that correctly identifies false instances. Little is known about the one-sided correlation which is correct on the true instances. A bound on this correlation will be derived using a novel monotone switching lemma for cliques, which only works in one direction. As a corollary, it is argued that computing the k-clique function on n vertex graphs requires the AND of at least $2^{n/4k}$ monotone, constant-depth, and polynomial-sized circuits, for sufficiently large values of k.

Then, depth 3 circuits computing the threshold function will be studied. A connection is proved between the block-based constructions and combinatorial decompositions. Under the assumption that a circuit satisfies this block structure, a matching size-lower bound is established. Additionally, a connection between problems in extremal combinatorics and the unconditional depth 3 complexity of THR is established, with various upper and lower bounds for the combinatorial question. Here, both the monotone and non-monotone settings are studied. It is argued that monotone depth 3 complexity of the threshold function is closely related to the generalized Turán problem for complete hypergraphs.

Next, new bounds will be established for this generalized Turán problem. Write $K_n^{(k)}$ for the complete k-graph on n vertices. It is proved that the maximum density of $K_g^{(k)}$ in large,

$K_r^{(k)}$-free k-graphs is at most $\prod_{m=k}^{g}\left(1-\frac{\binom{m-1}{k-1}}{\binom{r-1}{k-1}}\right)$. The proof uses techniques from the theory of flag algebras to derive linear relations between different densities. These relations can be combined using linear algebraic methods. Additionally, a simple sum-of-squares proof will be given for $\lim_{n\to\infty}\pi\left(n, K_4^{(3)}, K_5^{(3)}\right) = 3/8$.

The thesis contains the implementation of flag algebras in SAGE. Details about the code used to perform flag algebraic calculations will be discussed. The code permits calculations on any combinatorial theory, with a few common theories already implemented (graphs, hypergraphs, directed graphs, tournaments, graphs with ordered edges/vertices, graphs with colored edges). The implementation builds upon SAGE's powerful algebra base. This allows the manipulation of the flag algebra elements, providing easy and intuitive computation with them.

Finally, the thesis showcases a combinatorial setting for the satisfiability coding lemma. Under a structural assumption, a variant of the lemma is established that can be used inductively. Under the same assumption, tight constant-depth circuit size lower-bounds are proved for the parity function.

Contents

1 Introduction **1**

1.1 Circuits and Formulae 2

1.1.1 Switching Lemmas 2

1.1.2 Depth 3 Circuits 4

1.1.3 Bounds for Monotone Functions 5

1.2 Extremal Combinatorics 6

1.2.1 Generalized Turán Problems 7

1.3 Flag Algebras 8

2 Preliminaries **10**

2.1 Hypergraphs, Hypergraph Properties 11

2.2 Circuits and Formulae 13

2.2.1 Standard Form 15

2.2.2 Low-Depth Circuits and Formulae 16

2.3 The Flag Algebra Calculus 16

3 Contributions **22**

3.1 Top Gate of Circuits Computing CLIQUE 22

3.2 Depth 3 Circuits Computing THR 23

3.3 Hypergraph Turán Problems 25

3.4 Flag Algebra Calculus in SAGE 27

3.5 Satisfiability Coding Lemma Inductively 28

4 CLIQUE as an AND of Polynomial-Sized Monotone Constant-Depth Circuits **31**

4.1 Introduction . . . 31

4.2 Conventions . . . 34

4.3 Outline . . . 35

4.4 Proofs . . . 36

4.5 Concluding Remarks . . . 43

5 Depth 3 Circuits Computing THR **44**

5.1 Introduction . . . 44

5.1.1 Overview of the Results . . . 45

5.2 Block Restriction . . . 46

5.2.1 Size of $T_{\underline{b},r}$ . . . 49

5.3 Associated Hypergraph Problems . . . 50

5.3.1 Bounds for the Extremal Problems . . . 53

5.4 Concluding Remarks . . . 59

6 Generalized Turán Problem for Complete Hypergraphs **60**

6.1 Introduction . . . 60

6.1.1 Overview of the Result . . . 60

6.1.2 Outline of the Chapter . . . 63

6.1.3 Conventions . . . 63

6.2 Proof of Main Theorem . . . 63

6.3 Linear Density Relations . . . 66

6.4 The Associated Tridiagonal Matrix . . . 71

6.4.1 The inverse of $D_r^{(k)}$ . . . 73

6.4.2 The inverse of $D_r^{(k)} - \epsilon I$ . . . 76

6.5 $\pi\left(K_4^{(3)}, K_5^{(3)}\right) = 3/8$ . . . 78

6.6 Concluding Remarks . . . 80

6.6.1 Finding Squares 80

7 Flag Algebra Calculus in SAGE **82**

7.1 SDP and Flag Algebras 82

7.2 Implementation Details 83

7.2.1 `CombinatorialTheory` 84

7.2.2 `Flag` 85

7.2.3 `FlagAlgebra` 86

7.2.4 `FlagAlgebraElement` 86

7.3 Usage 87

7.3.1 Simple Optimization Problems 87

7.3.2 Evaluating Expressions 90

7.3.3 Explicit Squares 94

7.4 Constructing New Theories 98

8 Inductive Satisfiability Coding Lemma **101**

8.1 Introduction 101

8.1.1 Outline of the Chapter 102

8.2 Small Circuits computing Parity 103

8.3 Additional Notation 105

8.3.1 Sensitive Edges 106

8.3.2 Cones 107

8.4 Inductive Bound of Cones 110

8.4.1 Application to Parity 113

8.4.2 Proof of Lemma 65 114

8.5 Concluding Remarks 118

A Flag Algebra Code **130**

A.1 Combinatorial Theory, Flag Algebra, Flag Algebra Element 130

A.2 Flag 197

Chapter 1

Introduction

Finding fundamental limits on key complexity measures like time and space for general computational models is one of the core challenges in complexity theory. One such general model is the computation by Boolean circuits, an acyclic graph where each node computes an elementary function on its inputs (for example AND, OR). The computations represented by such Boolean circuits provide a fitting representation of the computational capabilities of physical hardware systems. Their combinatorial nature allows established mathematical tools to analyze questions regarding their limitations.

It is known that if an NP complete problem cannot be computed by polynomial-sized circuits, then NP $\neq$ P [AB09]. Thus, a potential approach for settling the P vs NP problem is by proving circuit lower bounds on NP problems. However, major barriers such as relativization [BGS75], natural proofs [RR97], and algebrization [AW08] show that current techniques cannot prove even modest complexity theory goals in general computational models. Indeed, a super-linear lower-bound on the size of logarithmic-depth circuits for an explicit NP problem is yet to be proven [Val]. As unconditional lower bounds for general computational models seem out of reach, the focus of complexity theory has shifted towards lower bounds for restricted computational models.

This thesis focuses on circuits and formulae (formally defined in Section 2.2). In particular, bounded depth and/or monotone circuits and formulae are considered. Such restricted circuit

models have provided a fruitful setting for unconditional complexity lower bounds.

1.1 Circuits and Formulae

1.1.1 Switching Lemmas

The first super-polynomial size bound for constant-depth circuits appeared in [FSS81] and independently in [Ajt83], showing that the parity function PAR, (returning the parity of ones in the input) cannot be expressed with constant-depth and polynomial-sized boolean circuits (for a formal definition see Chapter 2). This was refined in [Yao85] and [Hå86] developing the powerful switching method, establishing that depth d circuits computing parity require $2^{\Omega(n^{1/(d-1)})}$ size.

Switching lemmas provide a way to convert depth two circuits with top OR gates (DNF) into small depth 2 circuits with top AND gate (CNF) and vice-versa, by randomly fixing the value of some input variables. This provides a method to reduce the depth of a circuit. Håstad's probabilistic argument gives the following version of the switching lemma.

Theorem 1 ([Hå86] Main Lemma)**.** *Given a t-CNF f and $\rho \sim R_p$, the probability that $f|_\rho$ cannot be written as an s-DNF is at most α^s where α is the root of the equation*

$$\left(1 + \frac{4p}{(1+p)\alpha}\right)^t = \left(1 + \frac{2p}{(1+p)\alpha}\right)^t + 1.$$

Where R_p is a random restriction, leaving each input unassigned with probability p and setting the remaining inputs to 1 or 0 with probability $(1-p)/2$ each. For small enough p it is true that $\alpha < 5pt$, giving the probability bound $(5pt)^s$. By logical duality, note that the same probability bound applies to switching a t-DNF to an s-CNF. Note further that the function PAR, after such restriction reduces to another PAR (or its negation) on the remaining inputs, allowing the repeated usage of the lemma. Hence, an easy application of the switching lemma provides the following size lower bound for circuits computing PAR.

Theorem 2 ([Hå86] Theorem 1)**.** *Depth d circuits computing PAR on n input bits require*

size at least

$$2^{10^{-d/(d-1)}n^{1/(d-1)}}.$$

For the function CLIQUE_k (returning if a graph has a clique on k vertices), first Lynch [Lyn86] showed that among depth-d circuits, CLIQUE_k requires size at least $n^{\Omega\left(\sqrt{k/d^3}\right)}$. This was improved by Beame [Bea90] to $n^{\Omega\left(k/d^2\right)}$ and finally, for circuits with constant depth, Rossman showed [Ros08] the size lower bound $\Omega\left(n^{k/4}\right)$.

Beyond the application for cliques, the switching lemma has played a key role in circuit lower bounds for other functions and separation results. Using a block-based restriction, Håstad showed [Hå86] that the depth-d Sipser function SIPSER_d (returning the result of d layers of alternating AND and OR computations on blocks of the input) requires exponential-sized circuits on any depth smaller than d. In [Ros18], Rossman argued that depth-d formulae computing PAR require size $2^{\Omega\left((d-1)(n^{1/(d-1)}-1)\right)}$ using a sequential refinement of restrictions, that efficiently applies the switching lemma to formulae.

An alternative entropy-based proof of the switching lemma by Rossman [Ros17], which relies on the bound on the number of clauses instead of the bound on bottom fan-in, has been applied in various contexts. For example, it was used in the separation of monotone and non-monotone bounded-depth circuit classes [CO23].

The power of restrictions in analyzing unbounded fan-in Boolean circuits extends to other areas of complexity theory. One can obtain tight bounds on Fourier analytic properties of AC^0 functions using the switching lemma [LMN93, Tal17], providing better learnability algorithms.

The seminal work Pitassi, Beame and Impagliazzo [PBI93] extended the switching lemma to a setting where there is a great deal of dependency between the inputs. This established the first exponential lower bound on the size of bounded-depth Frege proofs for the pigeonhole principle. This bridge to proof complexity was followed with other successful applications in small-depth Frege systems, [PRST16, PRT22, Hå24] to name a few.

1.1.2 Depth 3 Circuits

The switching techniques illustrated above offer a simple method to analyze constant-depth circuits or formulae through an inductive process that reduces their depth. Naturally, this raises the question, whether this can be improved for the first non-trivial case; depth 3 circuits.

One possibility is to study the one-sided correlation between CNFs and a target function. Given that the size of the bottom layer in a circuit is small, it is possible to apply a small restriction where each clause turns into a k-clause, with k not too large. Therefore, it is often assumed that the depth 3 circuit has bottom fan-in bounded by some number k. Given a function $f : \{0,1\}^{[n]} \longrightarrow \{0,1\}$, finding any upper bound on

$$\pi_f^{k,h} = \max_{g:k\text{-CNF},g\Rightarrow f} \frac{\text{number of inputs where } g \text{ and } h \text{ agree}}{\text{number of inputs where } f \text{ and } h \text{ agree}} \tag{1.1}$$

provides a simple $1/\pi_f^{k,h}$ lower-bound for the size of the middle layer in depth 3 circuit with top OR gate, computing f, since at least $1/\pi_f^{k,h}$ many k-CNFs are required to cover the slice where f and h agree. Equation (1.1) can be easily studied with tools from extremal combinatorics. This line of approach was first investigated in [HJP95] for the function PAR and MAJ (returning if the majority of the inputs is true), with the bounds

$$\begin{aligned}
\pi_{\text{PAR}}^{k,h} &= O\left(e^{1/2k}\left(1-\frac{1}{k}\right)\right)^{n(1-\frac{1}{k})} \\
\pi_{\text{MAJ}}^{k,h'} &= O\left(\frac{\sqrt{k}}{2}\right)^{n},
\end{aligned}$$

where h and h' are symmetric functions, accepting inputs with weights from a suitable chosen subset of $[n]$. This provided size lower bounds for PAR and MAJ better than the ones obtained from switching lemmas. In particular, they found the following.

Theorem 3 (Theorem 3.3 and 3.5 from [HJP95]). *Any depth 3 circuit computing PAR requires size at least $2^{0.618\sqrt{n}}$. Any depth 3 circuit computing MAJ requires size at least $2^{0.849\sqrt{n}}$.*

This was substantially improved in [PPZ97] for PAR using a coding lemma for satisfying

assignments. Exploiting that any k-CNF appearing in the circuit must have isolated satisfying assignments (i.e. every satisfying assignment of every middle layer gate must have all its neighbours unsatisfying), they showed the tight bound

$$\pi_{\mathrm{PAR}}^{k,\{0,1\}^{[n]}} \leq 2^{-n/k}.$$

Another novelty of [PPZ97] is the application of the method with an average bottom fan-in, to get a precise lower bound on the total size of depth 3 circuits computing PAR. A further generalization in [PPSZ05] showed that the Bose–Chaudhuri–Hocquenghem (BCH) code [BRC60] has

$$\pi_{\mathrm{BCH}}^{k,\{0,1\}^{[n]}} \leq 2^{-cn/k},$$

with some $c > 1$, breaking the $2^{-n/k}$ barrier.

The coding lemma found great success in different settings. For example [Ama11] showed that among k-CNFs, PAR on k bits attains the highest sensitivity using the coding lemma. A generalization of the coding lemma in [MST22] gave a near optimal upper bound on the maximum number of prime implicants of a k-CNF. Recently [GPP+24] used a new local enumeration algorithm also based on the coding lemma to get the bound $\pi_{\mathrm{MAJ}}^{3,h} \lessapprox 0.799^n$ where h is a symmetric function, accepting inputs with weights less than $\lceil n/2 \rceil$.

1.1.3 Bounds for Monotone Functions

Every monotone circuit computes a monotone function, and conversely, every monotone function can be computed by a monotone circuit. Determining size bounds of monotone functions computable by monotone circuits is therefore a natural question. Many well-known functions such as CLIQUE and MAJ exhibit the monotone property.

Over the years, monotone Boolean circuits have been very well studied in computational complexity and continue to represent one of the few seemingly largest natural sub-classes of Boolean circuits for which exponential lower bounds have been obtained. This line of research began with an influential paper from Razborov [Raz85], showing monotone circuit complexity

bounds of $\Omega\left((n/\log^2 n)^k\right)$ for the CLIQUE_k function using monotone circuit approximators. This bound was subsequently improved by Alon and Boppana [AB87] to $\Omega\left((n/\log n)^k\right)$. Amano and Maruoka [AM] further extended the scope by deriving bounds for non-monotone circuits with a limited number (up to $(1/6)\log\log n$) of NOT gates.

For the majority function (MAJ), Boppana [Bop86] made an early breakthrough in 1986, proving that monotone circuits of bounded d depth require size $2^{\Omega\left(n^{1/(d-1)}\right)}$. Note that the best known upper bound construction achieves size around $2^{O\left(n^{1/(d-1)}(\log n)^{(d-2)/(d-1)}\right)}$. Although the switching lemma can derive the same asymptotic lower bound without the monotonicity assumption, nothing better is known in both the monotone and non-monotone settings despite subsequent efforts. Recently tight bounds in a restricted setting, where each gate in the middle layer can only query k variables was found in [LRT22]. Furthermore, [Ama23] established that negations can help reduce the size of the circuits computing MAJ, when the bottom fan-in is bounded by $k \in \{3, 4, 5\}$.

1.2 Extremal Combinatorics

Extremal combinatorics studies the relationship between local configurations in large structures. Equation (1.1) forms a natural subclass of the questions studied in extremal combinatorics, therefore can be analyzed with extremal combinatorial tools. The first question in this flavour was asked in 1907 by W. Mantel. As an exercise, W. Mantel determined the maximum number of edges an n vertex graph can have without a triangle [Man07]. This maximum is uniquely attained when the vertices are split into two groups of about equal sizes, and only the edges between the two groups are considered.

Independently, a landmark result by Turán determined the maximum number of edges a graph can have, without any copies of an r vertex complete graph exactly, with the unique graphs attaining the maximum.

Theorem 4 ([Pá41])**.** *$\pi\left(n, K_r^{(2)}\right)$ is uniquely attained at the balanced complete $(r-1)$-partite graph on n vertices.*

Here $\pi(n, H)$ means the maximum density of the edges in H-free structures. The limit, when $n \to \infty$ is simply denoted by $\pi(H)$. These values are formally defined in Chapter 2. Erdős and Stone found more generally the value $\pi(H)$ for all graphs H.

Theorem 5 ([ES46]). *Suppose H is a graph with chromatic number $\chi(H)$, then*

$$\pi(H) = 1 - \frac{1}{\chi(H) - 1}.$$

The corresponding question for k-graphs, when $k > 2$, is still open and seems to be much more difficult. There are sporadic results for various k-graphs, but no $\pi\left(K_r^{(k)}\right)$ value is known exactly. The best general upper bound comes from de Caen.

Theorem 6 ([dC83]).

$$\pi\left(n, K_r^{(k)}\right) \leq 1 - \left(1 + \frac{r-k}{n-r+1}\right) \frac{1}{\binom{r-1}{k-1}}.$$

For an extensive survey focusing on the $\pi\left(n, K_r^{(k)}\right)$ problem, with various lower and upper bounds, see [Sid95]. More recent coverage of the question with different k-graphs can be found in [Kee11].

1.2.1 Generalized Turán Problems

As a possible generalization of the Turán question, one can ask the maximum density of a given k-graph F, instead of the k-edges. Use $\pi(n, F; H)$ for the maximum density of F in H-free structures, and $\pi(F; H)$ for this value at the limit $n \to \infty$. For complete graphs, this was independently solved by Zykov [Zyk49] and Erdős [Erd62].

Theorem 7 ([Zyk49, Erd62]). *For $2 \leq g < r$ integers, the maximum number of K_g in K_r-free graphs (the value $\pi\left(n, K_g^{(2)}; K_r^{(2)}\right)$ formally) is uniquely attained at the balanced complete $(r-1)$-partite graph on n vertices.*

Asymptotically, this yields $\pi\left(K_g^{(2)}, K_r^{(2)}\right) = \prod_{m=2}^{g}\left(1 - \frac{m-1}{r-1}\right)$. The generalized Turán problem for graphs was systematically investigated by Alon and Shikhelman, obtaining a

result similar to Erdős-Stone.

Theorem 8 ([AS16]). *For any graph H, with chromatic number $\chi(H)$, the following holds*

$$\pi\left(K_g^{(2)}, H\right) = \prod_{m=2}^{g}\left(1 - \frac{m-1}{\chi(H)-1}\right).$$

Additionally, [AS16] investigates degenerate generalized Turán questions – the rate of convergence of $\pi(n, F, H)$ when $\pi(F, H) = 0$. [XZG21] finds various bounds for several degenerate generalized hypergraph Turán problems.

The generalized Turán problem for complete k-graphs corresponds to the separation of different layers of the boolean hypercube using a k-CNF. This idea appears for example in [Sid87]. [FGT22] gives new insights into the set of satisfying assignments of CNFs using a variant of the VC dimension. Bounds on this VC dimension variant turn out to be equivalent to a generalized Turán-type conjecture.

1.3 Flag Algebras

The theory of flag algebras [Raz07] provides a systematic approach to studying extremal combinatorial problems and the tools available for solving them. It gives a common ground for combinatorial ideas, by expressing them as linear operators, acting between flag algebras. Linearity means the different techniques can be easily combined with linear programming/linear algebra. The tools provided by flag algebras can be used to study for example Equation (1.1), improving the correlation bounds between circuits and various boolean functions.

A large part of the theory can be automated with state-of-the-art optimization algorithms, providing spectacular improvements in density bounds. There has been significant progress in the famous tetrahedron problem [dC83, CL99] with the previous best bound being $\pi\left(K_4^{(3)}\right) \leq \frac{3+\sqrt{7}}{12} < 0.59360$ while flag algebraic calculations improved it to the following bound:

Theorem 9 ([Raz10, FRV11, Bab11]). $\pi\left(K_4^{(3)}\right) \leq 0.56167$.

Note that the best known lower bound is $5/9 \leq \pi\left(K_4^{(3)}\right)$. For excluded $K_5^{(3)}$ the calculations give the following:

Theorem 10 ([Bab11])**.** $\pi\left(K_5^{(3)}\right) \leq 0.76954$

with best known lower bound $3/4 \leq \pi\left(K_5^{(3)}\right)$.

For a list of results provided by flag algebraic calculations, see [Raz10, FRV11]. The power of flag algebra has been illustrated in a wide range of other combinatorial questions [LP21, SS18]. Unfortunately, the computer-generated proofs lack insight and scale-ability compared to classical, hand-crafted arguments. They often only work in a small enough parameter range (for example, bounding $\pi(H)$ for H with at most 7 vertices). A survey by Razborov [Raz13] calls such applications plain.

Chapter 2

Preliminaries

The set of the first n integers will be denoted by $[n] = \{0, 1, ..., n-1\}$. For a set S, the collection of its subsets is $\{0,1\}^S := \{A : A \subseteq S\}$. There is a natural bijection between the power set and functions to $\{0,1\}$. Namely, $f : S \longrightarrow \{0,1\}$ corresponds to $F = f^{-1}(1)$. The two notions, function and collection of subsets of the input set, will be used interchangeably. Subsets of a given size k are denoted by $\binom{S}{k} := \{A : A \subseteq S, |A| = k\}$. Tuples (sets with an order) are represented in regular brackets, for example (a, b, c).

A function $f : \{0,1\}^{[n]} \longrightarrow \{0,1\}^{[m]}$ is monotone, if $x \leq y$ implies that $f(x) \leq f(y)$, under the standard partial order. Named functions are written with capital letters. Below is a list of the main named functions appearing in this thesis.

$$\text{AND} : \{0,1\}^S \longrightarrow \{0,1\} \qquad \text{AND}(A) \mapsto \begin{cases} 1 & \text{if } A = S \\ 0 & \text{otherwise} \end{cases}$$

$$\text{OR} : \{0,1\}^S \longrightarrow \{0,1\} \qquad \text{OR}(A) \mapsto \begin{cases} 0 & \text{if } A = \emptyset \\ 1 & \text{otherwise} \end{cases}$$

$$\text{NOT} : \{0,1\} \longrightarrow \{0,1\} \qquad \text{NOT} = \{0\}$$

$$\mathrm{PAR} : \{0,1\}^S \longrightarrow \{0,1\} \qquad \mathrm{PAR}(A) \mapsto |A| \mod 2$$

$$\mathrm{THR}_r : \{0,1\}^S \longrightarrow \{0,1\} \qquad \mathrm{THR}_r(A) \mapsto \begin{cases} 1 & \text{if } |A| > r \\ 0 & \text{otherwise} \end{cases}$$

$$\mathrm{CLIQUE}_r : \{0,1\}^{\binom{S}{2}} \longrightarrow \{0,1\} \qquad \mathrm{CLIQUE}_r(A) \mapsto \begin{cases} 1 & \text{if } \omega(A) \geq r \\ 0 & \text{otherwise} \end{cases}$$

Bold symbols represent random variables. The uniform distribution over a set S is represented by $\mathrm{Unif}(S)$. The probability that an event A happens is represented by $\mathbb{P}(A)$, while expected value of a random variable X is $\mathbb{E}(X)$.

2.1 Hypergraphs, Hypergraph Properties

The hypergraphs are identified with their edge sets; $G \subseteq \binom{V(G)}{k}$ is a k-graph with vertex set $V(G)$. In this thesis, size means $|V(G)|$. $K_n^{(k)}$ is the complete k-graph with size n. Equality means the hypergraphs are isomorphic; $G = H$ if there is a bijection $f : V(G) \longrightarrow V(H)$ that maps edges to edges and non-edges to non-edges. For $S \subseteq V(G)$ the induced sub-hypergraph can be represented as $G\restriction_S = G \cap \binom{S}{k}$. $\mathcal{H}_n^{(k)}$ is the collection of non-isomorphic k-graphs having n vertices, $\mathcal{H}^{(k)} = \bigcup_n \mathcal{H}_n^{(k)}$ is the collection of non-isomorphic k-graphs.

A hypergraph G has an induced copy of H if there is some $S \subseteq V(G)$ such that $H = G\restriction_S$, for short this is represented by $H \leq G$. Similarly, G has a copy of H if there is some S such that $H \subseteq G\restriction_S$ with $|S| = |V(H)|$. This is represented by $H \leq_s G$. The clique number of a hypergraph $\omega(G)$ is the largest r such that G has a copy of $K_r^{(k)}$.

A hypergraph property is any boolean function $P : \mathcal{H}^{(k)} \longrightarrow \{0,1\}$. A property is hereditary if a true instance remains true on all sub-instances, $\forall H : \big(P(G) \wedge H \leq G\big) \rightarrow P(H)$. The property is monotone hereditary if $\forall H : \big(P(G) \wedge H \leq_s G)\big) \rightarrow P(H)$.

The density of H in G is defined to be $d(H,G) = \mathbb{P}[G\restriction_{\mathbf{S}} \simeq H]$ where $\mathbf{S} \sim \text{Unif}\binom{V(G)}{|H|}$. Notice that this is the induced density, not the classical inclusion. Write

$$d_s(H,G) = \mathbb{P}[G\restriction_{\mathbf{S}} \simeq F, H \subseteq F]$$

for the classical inclusion with the same $\mathbf{S} \sim \text{Unif}\binom{V(G)}{|H|}$. Note when H is a complete hypergraph, the two notions are equivalent.

For a property P write $\pi(n,F;P) = \max\{d(F,G) : G \in \mathcal{H}_n, \quad P(G)\}$ the maximum number of copies of F in n size hypergraphs satisfying P. When P is hereditary, the limit as n goes to infinity is well-defined, and is denoted by $\pi(F;P) := \lim_{n\to\infty} \pi(n,F;P)$. For short, when F is omitted, then F is a single k-edge.

For a set of hypergraphs $\underline{H} = \{H_0, H_1, ...\}$, a simple hereditary property can be constructed by checking if any of the H_i appears as an induced copy.

$$P_{\underline{H}}(G) := \bigwedge_{H\in\underline{H}} \neg H \leq G.$$

As an abuse of notation, the property and the list of hypergraphs is identified. Furthermore, when the list has a single element, the element and the list is identified. In particular, the generalized Turán problem is to determine the value

$$\pi(n,F;H) = \max\{d_s(F,G) : G \in \mathcal{H}_n, \quad d_s(H,G) = 0\}$$

for various F, H hypergraphs.

Graph is the same as 2-graph, in particular has no loops or multiple edges. A directed graph (digraph) D is identified by the edge set $D \subseteq V(D) \times V(D)$. The edge $\{u,v\}$ from a graph is shortened to uv, and for directed graphs (u,v) is shortened to $\overrightarrow{uv}$.

In a graph G, the neighbours of v are $N_G(v) := \{u : uv \in G\}$. When the graph is understood, then the subscript G is dropped. Similarly, in a directed graph D, the in-neighbours of $v \in V(D)$ are denoted by $N_{D,\text{in}}(v) := \{u : \overrightarrow{uv} \in E(D)\}$, and the out-neighbours

$N_{D,\text{out}}(v) := \{u : \overrightarrow{vu} \in E(D)\}$, with the D dropped when clear from context.

A path in a graph G is any sequence of distinct ($v_i \neq v_j$) vertices $v_0, v_1, ..., v_{l-1}$ such that $v_i v_{i+1} \in G$. For digraphs D, analogously a path is a sequence $v_0, v_1, ..., v_{l-1}$ of distinct vertices, with $\overrightarrow{v_i v_{i+1}} \in D$. A cycle is a path with an additional $v_l = v_0$ vertex connected as expected $v_{l-1} v_l \in G$. A graph without a cycle is a forest. Every digraph has a corresponding graph, where the edge orientation is removed. In particular $V(D) \times V(D)$ is mapped to $\binom{V(D)}{2}$ using the simple map $(u, v) \mapsto \{u, v\}$.

2.2 Circuits and Formulae

In this thesis, circuits are tuples $\mathcal{C} := (S, D, \mathcal{L}, \mathcal{N}, \mathcal{O})$, where S is the set of inputs, D is a directed acyclic graph. Elements of $V(D)$ are called gates, instead of vertices, as per tradition. $\mathcal{L} : V(D) \longrightarrow S \cup \{\wedge, \vee\}$ is a labelling on the gates, $\mathcal{N} : E(D) \longrightarrow \{1, -1\}$ is an indicator for negated edges and $\mathcal{O} \subseteq V(D)$ is the collection of output gates. With the abuse of notation, write $V(\mathcal{C}) := V(D)$ and $E(\mathcal{C}) := E(D)$ for the gates and edges in the circuit. For $v \in V(\mathcal{C})$, when $\mathcal{L}(v) \in S$, then v is called an input gate, and is expected to have no in-neighbours.

For $v \in V(\mathcal{C})$ the depth $d(v)$ is defined to be the longest path from the input gates to v. The depth of the circuit is $d(\mathcal{C}) := \max\big(d(v) : v \in \mathcal{O}\big)$. A layer is a collection of gates with a given depth i and is denoted by $V_i(\mathcal{C}) := \{v \in V(\mathcal{C}) : d(v) = i\}$. Since input gates have no in-neighbours, their depth is 0. For a simpler notation, write $w(v) := \big|N_{\text{in}}(v)\big|$, which is traditionally called the fan-in of the gate. The size of the circuit is defined to be

$$s(\mathcal{C}) := \Big|\big\{v : v \in V(\mathcal{C}),\ \mathcal{L}(v) \in \{\wedge, \vee\}\big\}\Big|.$$

Each $v \in V(\mathcal{C})$ and $e \in E(\mathcal{C})$ computes a function $f : \{0,1\}^S \longrightarrow \{0,1\}$, write $f_v, f_e \subseteq \{0,1\}^{\{0,1\}^S}$ for these functions. f_v and f_e are defined recursively:

- for $v \in V(\mathcal{C})$ an input gate, the function is

$$f_v := \{A \subseteq S : \mathcal{L}(v) \in A\}$$

- for $v \in V(\mathcal{C})$ with $\mathcal{L}(v) = \wedge$ the function is

$$f_v := \bigcap_{u \in N_{\text{in}}(v)} f_{\overrightarrow{uv}}$$

- for $v \in V(\mathcal{C})$ with $\mathcal{L}(v) = \vee$ the function is

$$f_v := \bigcup_{u \in N_{\text{in}}(v)} f_{\overrightarrow{uv}}$$

- for $\overrightarrow{uv} \in E(\mathcal{C})$ with $\mathcal{N}(\overrightarrow{uv}) = 1$ the function is

$$f_{\overrightarrow{uv}} := f_u$$

- for $\overrightarrow{uv} \in E(\mathcal{C})$ with $\mathcal{N}(\overrightarrow{uv}) = -1$ the function is

$$f_{\overrightarrow{uv}} := \{0,1\}^{\{0,1\}^S} \setminus f_u.$$

The entire circuit $\mathcal{C} = (S, D, \mathcal{L}, \mathcal{N}, \mathcal{O})$ computes the function $f_{\mathcal{C}} : \{0,1\}^S \to \{0,1\}^{\mathcal{O}}$, which is the combination of the functions computed by the output gates $f_{\mathcal{C}}(X) := (f_v(X) : v \in \mathcal{O})$.

Formulae are circuits with the additional restriction that the digraph $D \restriction_{V(D) \setminus S}$ viewed as a regular graph is a forest.

For a given circuit $\mathcal{C}$ with a single output gate, a satisfying assignment is any input X such that $f_{\mathcal{C}}(X) = 1$.

The restriction $\rho : A \longrightarrow \{0, 1, \star\}$ transforms any Boolean function $f : \{0,1\}^A \longrightarrow \{0,1\}$ into $f|_\rho : \{0,1\}^{\rho^{-1}(\star)} \longrightarrow \{0,1\}$ such that it computes $f|_\rho(B) \mapsto f(B \cup \rho^{-1}(1))$ for all $B \subseteq \rho^{-1}(\star)$. By independently picking the value of $\rho(i)$, a simple random restriction can be

generated. R_p is a distribution on restrictions such that $\rho \sim R_p$ has

$$\rho(i) \mapsto \begin{cases} \star & \text{with probability } p \\ 1 & \text{with probability } (1-p)\frac{1}{2} \\ 0 & \text{with probability } (1-p)\frac{1}{2} \end{cases}$$

for each i independently in the input set.

2.2.1 Standard Form

A circuit is said to be in standard form if:

- for every $\overrightarrow{uv} \in E(\mathcal{C})$ with $\mathcal{L}(u) \notin S$ it follows that $\mathcal{N}(\overrightarrow{uv}) = 1$
- for $i > 0$, every $u, v \in V_i(\mathcal{C})$ has $\mathcal{L}(u) = \mathcal{L}(v)$
- for every $\overrightarrow{uv} \in E(\mathcal{C})$ it holds that $d(v) = d(u) + 1$
- every $v \in \mathcal{O}$ has the same $d(v) = d(\mathcal{C})$ depth.

In words, the $\wedge$ and $\vee$ gates alternate, and negation edges only appear between input gates and gates in the first layer. In addition, every path from the input gates to a given gate has the same length and this length agrees with the depth of the gate. Since a fixed layer can only contain one type of gate label, write $\mathcal{L}_i(\mathcal{C})$ for that label.

Write $s_i(\mathcal{C}) := \left|V_i(\mathcal{C})\right|$ for the size of layer i and $s_{\max}(\mathcal{C}) := \max\limits_{0<i\leq d(\mathcal{C})} s_i(\mathcal{C})$ for the size of the largest layer. Note that

$$s_i(\mathcal{C}) \leq s_{\max}(\mathcal{C}) \leq s(\mathcal{C}) = \sum_{0<i} s_i(\mathcal{C}) \leq d(\mathcal{C})\ s_{\max}(\mathcal{C}).$$

The following is a well-known reduction.

Theorem 11. *For every circuit $\mathcal{C}_1$, there is a circuit $\mathcal{C}_2$ in standard form computing the same function with $s_{\max}(\mathcal{C}_2) \leq 2s(\mathcal{C}_1)$ and $d(\mathcal{C}_2) = d(\mathcal{C}_1)$.*

Since the size difference is at most a $2d$ factor, from now on, every circuit is assumed to be in standard form.

Circuits/formulae without negation gates are called monotone circuits/formulae.

2.2.2 Low-Depth Circuits and Formulae

While depth 0 circuits/formulae are the input gates, the edges coming from the inputs could encode either the input or the negation of the input. They are called (positive, negative) literals.

A single output depth 1 circuit/formula (in the standard form) therefore computes the AND or the OR of literals. They are called (AND, OR) clauses. A clause with k fan-in is simply called a k-clause. A clause with only positive literals is called a monotone clause. A clause with at most k positive literals is called a k^- clause.

A single output depth 2 circuit/formula computing the AND of OR clauses is called a CNF, while a depth 2 circuit computing the OR of AND clauses is a DNF. In addition, the k-, k^-- and monotone prefixes indicate that the property applies to all clauses. For example a k^--DNF is a DNF with only k^- clauses.

Similarly, depth 3 circuits could have AND or OR output gates. They are referred to as top gates. The middle layer is $V_2(\mathcal{C})$, and the size of the middle layer of the circuit is $s_2(\mathcal{C})$.

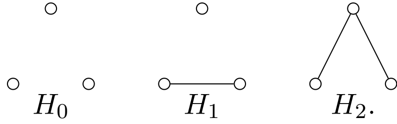
In general, Π_d-circuit/formula means the class of depth d circuits/formulae with top AND gate, while Σ_d represents the class of depth d circuits/formulae with top OR gate. The superscripts restrict the bottom layer and the m prefix indicates a monotone class, for example $m\Pi_d^k$-formula means the class of monotone depth d formulae with bottom fan-in bounded by k.

2.3 The Flag Algebra Calculus

The theory of flag algebras was introduced in [Raz07]. This section introduces the definitions and concepts pertinent for this thesis. The exposition of Mantel's theorem with a flag algebraic upper bound was inspired by [SFS16], and will provide a good setting for the definitions.

Theorem 12 (Asymptotic upper bound for Mantel's theorem)**.** $\pi\left(K_3^{(2)}\right) \leq 1/2$

Proof. The (2) superscript will be omitted in this proof. Consider a large enough graph G without K_3, and pick 3 of its vertices uniformly randomly. The possible induced graphs those 3 vertices can span are:



Writing $h_i = d(H_i, G)$ it is clearly true that $h_0 + h_1 + h_2 = 1$, and $0 \leq h_i \leq 1$. Choosing two vertices uniformly randomly, one gets either an induced edge or an empty pair:

Similarly to the above, writing $e_i = d(E_i, G)$, it holds that $e_0 + e_1 = 1$ while $0 \leq e_i \leq 1$. By the law of total probability, getting E_1 conditioning on a randomly chosen triple gives the identity

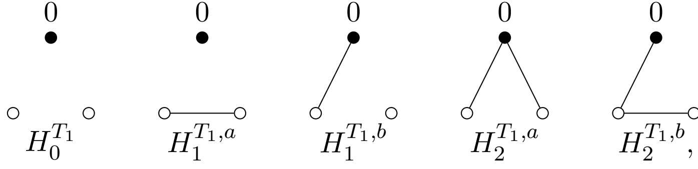
$$e_1 = \frac{1}{3}h_1 + \frac{2}{3}h_2, \tag{2.1}$$

and similarly, for the non-edge the identity

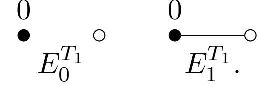
$$e_0 = h_0 + \frac{2}{3}h_1 + \frac{1}{3}h_2. \tag{2.2}$$

Using these simple relations, it is easy to deduce that $e_1 \leq 2/3$ in any triangle-free graph, therefore $\pi\left(K_3^{(2)}\right) \leq 2/3$. Note that this bound follows from the enumeration of the possible triangle-free graphs of size 3, and by noting that in a large graph, each induced subgraph of size 3 can be just as good as the optimal triangle-free 3 size graph. The optimal triangle-free 3 sized graph has 2 edges, giving the bound 2/3. In general one can get slightly better bounds by finding the optimal structure for larger and larger graphs. For example, the optimal 5 sized graph without triangle has 6 edges, giving a slightly better 3/5 upper bound.

Further constraints on the e_i, h_i values could yield better bounds on the density. Suppose one vertex in G is labelled, say with label 0. The possible non-isomorphic three vertex subgraphs containing the labelled vertex then are:



And the two vertex subgraphs containing the labelled vertex can be:



Given a fixed G^{T_1} graph, write the densities of each labelled 2 and 3 vertex substructure with the lower-case variant as above. Then

$$\begin{aligned} 0 \le h_i^{T_1} \le 1, &\qquad \sum_i h_i^{T_1} = 1 \\ 0 \le e_i^{T_1} \le 1, &\qquad \sum_i e_i^{T_1} = 1 \\ e_0 = h_0^{T_1} + h_1^{T_1,a} + \frac{1}{2}h_1^{T_1,b} + \frac{1}{2}h_2^{T_1,b} \\ e_1 = \frac{1}{2}h_1^{T_1,b} + h_2^{T_1,a} + \frac{1}{2}h_2^{T_1,b}. \end{aligned} \tag{2.3}$$

The first two lines simply indicated that the probabilities are between 0 and 1, and that they sum to 1. The last two lines follow from the chain-rule, just like in Equation (2.1).

In addition, if G^{T_1} is large enough, choosing uniformly randomly a three vertex substructure, with one vertex labelled, is approximately the same as choosing independently two copies of a two vertex substructure, with one labelled vertex. Since the probability that the same copy is selected is negligible in large structures, it follows that:

$$\begin{aligned} \left(e_0^{T_1}\right)^2 \approx h_0^{T_1} + h_1^{T_1,a} \\ \left(e_1^{T_1}\right)^2 \approx h_2^{T_1,a} \\ e_1^{T_1} e_0^{T_1} \approx \frac{1}{2}h_1^{T_1,b} + \frac{1}{2}h_2^{T_1,b}. \end{aligned} \tag{2.4}$$

These relations only hold in labelled graphs. Averaging over all possible choices of the vertex T_1, the following is not hard to verify:

$$
\begin{aligned}
\mathbb{E}_{T_1\sim\mathrm{Unif}(V(G))}\ e_0^{T_1} &= e_0 \\
\mathbb{E}_{T_1\sim\mathrm{Unif}(V(G))}\ e_1^{T_1} &= e_1 \\
\mathbb{E}_{T_1\sim\mathrm{Unif}(V(G))}\ h_0^{T_1} &= h_0 \\
\mathbb{E}_{T_1\sim\mathrm{Unif}(V(G))}\ h_1^{T_1,a} &= \frac{1}{3}h_1 \\
\mathbb{E}_{T_1\sim\mathrm{Unif}(V(G))}\ h_1^{T_1,b} &= \frac{2}{3}h_1 \\
\mathbb{E}_{T_1\sim\mathrm{Unif}(V(G))}\ h_2^{T_1,a} &= \frac{1}{3}h_2 \\
\mathbb{E}_{T_1\sim\mathrm{Unif}(V(G))}\ h_2^{T_1,b} &= \frac{2}{3}h_2.
\end{aligned}
\tag{2.5}
$$

Since the matrix

$$
M^{T_1} = \begin{pmatrix} \left(e_0^{T_1}\right)^2 & e_1^{T_1}e_0^{T_1} \\ e_1^{T_1}e_0^{T_1} & \left(e_1^{T_1}\right)^2 \end{pmatrix} \approx \begin{pmatrix} h_0^{T_1} + h_1^{T_1,a} & \frac{1}{2}h_1^{T_1,b} + \frac{1}{2}h_2^{T_1,a} \\ \frac{1}{2}h_1^{T_1,b} + \frac{1}{2}h_2^{T_1,a} & h_2^{T_1,a} \end{pmatrix}
$$

is positive semidefinite, and positive semi-definiteness is preserved under convex combinations, after averaging, the resulting matrix

$$
\mathbb{E}_{T_1\sim\mathrm{Unif}(V(G))}\ M^{T_1} \approx \begin{pmatrix} h_0 + \frac{1}{3}h_1 & \frac{1}{3}h_1 + \frac{1}{3}h_2 \\ \frac{1}{3}h_1 + \frac{1}{3}h_2 & \frac{1}{3}h_2 \end{pmatrix} \tag{2.6}
$$

must be positive semidefinite too. This provides an additional constraint the values h_0, h_1, h_2, e_0, e_1 must satisfy. Under the conditions Equation (2.1), Equation (2.2), Equation (2.6), the maximum value of e_1 appears at $1/2$ giving the exact upper bound for the asymptotic Mantel's theorem, that $\pi\left(K_3^{(2)}\right) \leq 1/2$. □

The general theory of flag algebras places the above ideas and steps in a common setting. The entire calculation can be performed in any theory with only relational symbols and universal axioms. Write $\mathcal{H}$ for the collection of finite models in the theory and $\mathcal{H}_n$ for the models with size n. The above example assumed a given large-enough graph. A more rigorous

setting performs these calculations in a limit structure, on a sequence of models $G_n \in \mathcal{H}_n$ with increasing size, where these densities converge. The flag algebra is an algebra whose elements are the $\lim_{n\to\infty} d(F, -)$ objects for various F, and their linear combinations. In particular, over the base ring B (usually $\mathbb{Q}$ or $\mathbb{R}$), the free vector space is $B^{\mathcal{H}}$, where the atom F also means the object $\lim_{n\to\infty} d(F, -)$. Here

$$\sum b_i F_i \in B^{\mathcal{H}}$$

corresponds to the expression

$$\lim_{n\to\infty} \sum b_i d(F_i, -),$$

or evaluated on a fixed sequence of structures G_n,

$$\lim_{n\to\infty} \sum_i b_i d(F_i, G_n).$$

As observed in Equation (2.1) and Equation (2.2), the chain rule provides expressions in $B^{\mathcal{H}}$ that evaluate to 0 on all large enough structures. Writing

$$K = \left\{ \sum b_i F_i \in B^{\mathcal{H}} : \forall G, \text{ large enough, } \sum_i b_i d(F_i, G) = 0 \right\}$$

for this kernel, the quotient space is the flag algebra

$$\mathcal{A} = B^{\mathcal{H}}/K.$$

The double-counting used in Equation (2.4) corresponds to a well-defined multiplication in the algebra. Note that the elements of this algebra are all symbolic. The evaluation of $\lim_{n\to\infty} d(F, -)$ elements on any increasing sequence G_n, namely $\lim_{n\to\infty} d(F, G_n)$, defines an algebra homomorphism from the flag algebra to the reals $\mathbb{R}$.

The labelled elements in the above example are called types in the literature. As demonstrated in Equation (2.3), the labelled structures also form a linear space where the linear

relations can be removed with a quotient. A type is simply a labelled subset of any hypergraph. Types are equal if they are isomorphic with respect to the ordering. If T is a type, then the collection of models with T type is $\mathcal{H}^T$, or $\mathcal{H}_n^T$ when the size is specified. Then one can similarly construct the kernel and the quotient space to define the flag algebra with type $\mathcal{A}^T$.

The averaging performed at the step Equation (2.5) is a mapping between the flag algebras with different types. This mapping preserves linearity, as demonstrated in Equation (2.5). This mapping loses the multiplicative property, therefore is not an algebra homomorphism, but preserves the positive semi-definiteness of the matrices. The averaging operator will be denoted by $\left[\!\left[F^T\right]\!\right]_T \in \mathcal{A}$, in particular, it is a linear map (but not an algebra homomorphism)

$$[\![-]\!]_T : \mathcal{A}^T \longrightarrow \mathcal{A}.$$

As mentioned above, in this thesis, the hypergraphs will represent the corresponding flag algebra elements, i.e. the hypergraph F will denote the corresponding $\lim_{n\to\infty} d(F,-)$ element without any type. The types will be indicated as a superscript, as in the above examples $E_0^{T_1}$. $T_n^{(k)}$ is the complete type on the language of k-graphs with n vertices and all k-uniform edges. In particular $K_n^{(k),T_m^{(k)}}$ is the unique complete flag on n vertices with a type having m vertices. For a type T, T also represents the flag with type T and no extra vertices/edges.

Chapter 3

Contributions

3.1 Top Gate of Circuits Computing CLIQUE

The simplest circuit expressing CLIQUE, is an OR of monomials, one for each possible clique. The top gate has fan-in $\binom{n}{k} \approx n^k$. The complexity of CLIQUE_k is well studied. Under a constant-depth restriction, Lynch [Lyn86] established the first superpolynomial size lower-bound. This was subsequently refined in [Bea90] and finally in [Ros08], establishing an $\Omega\left(n^{k/4}\right)$ size lower-bound. Regarding monotone circuits, first Razborov [Raz85] proved a super polynomial size lower bound using the sunflower lemma from Erdős and Rado [ER60]. A long line of improvements [AB87, CKR20] established the current best $n^{\Omega(k)}$ size lower-bound when $k \leq n^{1/3-o(1)}$.

The trivial circuit construction calculating CLIQUE directly translates to a monotone monadic existential second order formula mMSO∃. Showing that there is no universal second order sentence (SO∀) for CLIQUE would separate NP and co-NP.

Dually to the trivial construction, one can express CLIQUE as monotone AND of monomials, one for each maximal graph without k-clique. This gives an $\exp\left(\Omega\left((n-k)^2\right)\right)$ sized monotone, depth-2 circuit, with top AND gate, computing the k-clique function. Showing that the AND of at least $2^{\omega(n)}$ many monotone, constant-depth, polynomial-sized circuits are required to express CLIQUE would show that CLIQUE is not in MSO∀. Chapter 4 makes a partial progress towards this direction, giving a bound on the number of monotone

constant-depth polynomial sized circuits needed, such that their AND expresses CLIQUE.

Theorem 13. *For any c_d, c_s constants, large enough n and $\log(n)^{c_d+6} < k$, one cannot have less than $2^{n/4k}$ many monotone circuits $\{f_j\}$ on $\binom{n}{2}$ input bits, each with depth and size bounded by c_d and n^{c_s} respectively, such that $\bigwedge_j f_j$ computes the k-clique function.*

The bound looks odd (and surprisingly strong for small k) compared to classical circuit bounds for CLIQUE, due to the restricted dual setting. This shows that OR gates are much better at expressing CLIQUE than AND gates in this sense.

The main novelty of the proof is an approximation of a monotone CNF circuit having larger fan-in, by a monotone DNF circuit having a small fan-in, where this approximation is correct on most of the relevant cliques.

3.2 Depth 3 Circuits Computing THR

Establishing a $2^{\omega(\sqrt{n})}$ depth 3 circuit size lower-bound for an explicit function remains one of the biggest challenges in theoretical computer science.

Although it appears straightforward, still a significant gap between the upper and lower bounds on the size of depth 3 circuits persists. A counting argument shows that a random function on n variables requires depth 3 circuits of size $\Theta\left(2^{n/2}\right)$ [Dan96, Ser19]. However, the best-known lower bound on the size of a depth 3 circuit for an explicit Boolean function is $2^{\Omega(\sqrt{n})}$ [Hå86, HJP95, PPZ97]. A $2^{\omega(\sqrt{n})}$ lower bound on the size of depth 3 circuit has been unresolved for over 30 years. A suitable candidate is the majority function $\text{MAJ} = \text{THR}_{n/2}$. It is conjectured that any depth 3 circuit computing MAJ requires $2^{\Theta(\sqrt{n \log_2 n})}$ size, following from a block construction. However, the best known depth 3 circuit size lower bound for MAJ is only $2^{O(\sqrt{n})}$ [HJP95].

In Chapter 5, depth 3 circuits computing the THR_r function with a top AND gate and k-bounded bottom fan-in will be considered. Due to the restricted bottom fan-in setting, bounds on the size of the middle layer will be derived for circuits computing the THR_r function. As THR_r is a monotone function, Chapter 5 additionally explores the question restricted to

monotone depth 3 and bounded bottom fan-in circuits.

The most efficient known method for computing the THR_r function, in terms of the size of the middle layer, employs a block construction approach. For the precise statement, first the set of decompositions is defined.

Definition 14. *For a sequence* $\underline{b} = (b_0, b_1, ..., b_{l-1})$ *and* $0 \le r \le \sum_{j\in[l]} b_j$*, write*

$$T_{\underline{b},r} = \left\{ (t_0, t_1, ..., t_{l-1}) \ : \ 0 \le t_j \le b_j, \ \sum_{j\in[l]} t_j = r \right\}$$

for the set of decompositions.

The quantity $\left|T_{\underline{b},r}\right|$ is exactly the decompositions of the number r into l many terms, each bounded by the corresponding b_j value.

Theorem 15. *Suppose the* $\underline{B} = (B_0, B_1, ..., B_{l-1})$ *sets partition the input bits* $[n]$*, with corresponding sizes* $\underline{b} = \big(|B_0|,\ |B_1|,\ ...,\ |B_{l-1}|\big)$*. Then there is a* $m\Pi_3^k$*-formula, with* $k = \max(\underline{b})$ *whose middle layer has size* $\left|T_{\underline{b},r}\right|$*, computing* THR_r*.*

Additionally, Section 5.2 argues that any circuit computing THR with a block structure must have the number of gates in the middle layer equal to the above construction.

Theorem 16. *Suppose the* $\underline{B} = (B_0, B_1, ..., B_{l-1})$ *sets partition the input bits* $[n]$*, with corresponding sizes* $\underline{b} = \big(|B_0|,\ |B_1|,\ ...,\ |B_{l-1}|\big)$*. If a* Π_3^k*-circuit*

$$\bigwedge_{i\in[m]} \bigvee_{C\in D_i} \bigwedge_{l\in C} l$$

computes THR_r*, where for each term* $C \in D_i$ *there is some block* B_j*, such that* C *only contains literals from* B_j*, then the number of gates in the middle layer is at least* $m \ge \left|T_{\underline{b},r}\right|$*.*

Section 5.3 connects the unconditional question with various extremal hypergraph problems. A simple connection is established that relates the one-sided correlation between Π_3^{k-} and THR_r, with a question in extremal combinatorics (the Turán number of property $P_{k,r}^{(g)}$).

This associated hypergraph question is completely resolved for $k = 2$, and a general bound is obtained for larger k.

Theorem 17. *For $k = 2$, suppose $n = ar + b$ with a, b integers and $0 \leq b < r$ (division with remainder), then*

$$\pi\left(n; P_{2,r}^{(r)}\right) = \frac{a^{r-b}(a+1)^b}{\binom{n}{r}}.$$

For general k, it holds that

$$\pi\left(n; P_{k,r+1}^{(r)}\right) \leq \frac{k^{r/(k-1)}}{\binom{rk/(k-1)}{r}}.$$

The section also proves a similar connection between the monotone variant of the problem and a generalized hypergraph Turán question (the generalized Turán number $\pi\left(K_r^{(k)}; K_{r+1}^{(k)}\right)$). The best-known bound for this generalized Turán question is established in the following Chapter 6.

3.3 Hypergraph Turán Problems

It is a long-standing open problem in Extremal Combinatorics to develop some understanding of the hypergraph Turán questions. Chapter 6 investigates the generalized hypergraph Turán problem. Here the maximum number of complete k-graphs on g vertices is studied, under the assumption that there are no complete k-graph on r vertices. Formally this means the values $\pi\left(n, K_g^{(k)}, K_r^{(k)}\right)$ are considered.

One of the goals of Chapter 6 is to show that the powerful flag algebra method can be performed by hand, resulting in a general and scaleable theorem. The main ideas and proof steps, therefore, correspond with the typical application of flag algebra and were heavily inspired by it. However, instead of computer generated inequalities, an infinite family of inequalities is established, providing a bound for a parameter range outside the limits of the computer generated proofs. The main contribution of the chapter is:

Theorem 18. *For integers* $1 < k \le g < r$ *and any* $n > (r-1)\left(1 + \left(\frac{(r-k)}{k-1}\right)^2\right)$,

$$\pi\left(n, K_g^{(k)}, K_r^{(k)}\right) \le$$
$$\le \left(1 + \frac{(r-1)(r-k)^2}{(k-1)^2 n - (r-1)\left(2k^2 - 2k(r+1) + r^2 + 1\right)}\right) \prod_{m=k}^{g} \left(1 - \frac{\binom{m-1}{k-1}}{\binom{r-1}{k-1}}\right).$$

Note this means asymptotically that

Theorem 19. *For integers* $1 < k \le g < r$

$$\pi\left(K_g^{(k)}, K_r^{(k)}\right) \le \prod_{m=k}^{g} \left(1 - \frac{\binom{m-1}{k-1}}{\binom{r-1}{k-1}}\right).$$

The asymptotic bound is known to be tight when $k = 2$, with the matching, balanced $(r-1)$-partite construction. Additionally, it agrees with the best-known general hypergraph Turán bound by de Caen [dC83] which is conjectured to not be tight. The main tool used in the proof of Theorem 18 is:

Lemma 20. *For all* $0 \le x$ *and integers* $k \le m < n$, *if* G *is an* n *vertex* k*-graph then*

$$\begin{aligned} 0 \ge \quad & \left(-\frac{1 - \frac{k-1}{m}}{x}\right) && d\left(K_{m+1}^{(k)}, G\right) && + \\ & \left(2 - \frac{k-1}{mx} - \frac{1}{(n-m)x}\right) && d\left(K_m^{(k)}, G\right) && + \\ & (-x) && d\left(K_{m-1}^{(k)}, G\right). \end{aligned}$$

Note that the densities $d\left(K_m^{(k)}, G\right)$ only appear linearly in the expression. For different k, g, r parameters, a convex combination of the expressions appearing in Lemma 20, with suitable x, m values substituted, yields Theorem 18. As a comparison, [dC83] utilizes similar ideas, but with a more complicated (non-linear) expression,

$$f_{m+1} \ge \frac{m^2 f_m}{(m-k+1)(n-m)} \left(\frac{f_m(n-m+1)}{f_{m-1} m} - \frac{(k-1)(n-m)+m}{m^2}\right)$$

where $f_m = d\left(K_m^{(k)}, G\right)$ for short.

In addition, the surprising exact result $\pi\left(K_4^{(3)}, K_4^{(3)}\right) = 3/8$ is also proved in Chapter 6.

3.4 Flag Algebra Calculus in SAGE

Chapter 7 contains details about the code used to perform flag algebraic calculations. The code permits calculations on any combinatorial theory, with a few common theories already implemented (graphs, hypergraphs, directed graphs, tournaments, graphs with ordered edges/vertices, graphs with colored edges). The main components are explained with detailed examples for each, also highlighting the generality of the software.

The software is able to evaluate flag algebraic expressions. This example calculates a symbolic representation of the relationship showcased in Equation (2.4) and Equation (2.5).

```
pointed_edge = GraphTheory(2, ftype=[0], edges=[[0, 1]])
squared_pointed_edge = pointed_edge * pointed_edge
projected_squared_pointed_edge = squared_pointed_edge.project()
print(squared_pointed_edge, "\n\n", projected_squared_pointed_edge)
```

```
Flag Algebra Element over Rational Field
0 - Flag on 3 points, ftype from [0] with edges=[]
0 - Flag on 3 points, ftype from [0] with edges=[[0, 2]]
0 - Flag on 3 points, ftype from [1] with edges=[[0, 2]]
0 - Flag on 3 points, ftype from [0] with edges=[[0, 2], [1, 2]]
1 - Flag on 3 points, ftype from [2] with edges=[[0, 2], [1, 2]]
1 - Flag on 3 points, ftype from [0] with edges=[[0, 1], [0, 2], [1, 2]]

Flag Algebra Element over Rational Field
0   - Flag on 3 points, ftype from [] with edges=[]
0   - Flag on 3 points, ftype from [] with edges=[[0, 2]]
1/3 - Flag on 3 points, ftype from [] with edges=[[0, 2], [1, 2]]
1   - Flag on 3 points, ftype from [] with edges=[[0, 1], [0, 2], [1, 2]]
```

The software is also able to upper bound Turán type questions automatically. For example a tight (up to machine rounding errors) upper bound for Mantel's theorem can be produced with the following:

```python
triangle = GraphTheory(3, edges=[[0, 1], [0, 2], [1, 2]])
GraphTheory.exclude(triangle)
edge = GraphTheory(2, edges=[[0, 1]])
print(GraphTheory.optimize_problem(edge, 3))
```

```
...
0.5000000004501909
```

A similar software, called flagmatic [FRV11], was previously available. Due to the lack of maintenance it became outdated and unable to run. In addition, this new implementation allows a more direct flag algebraic calculus, and builds on top of the SAGE mathematical framework. A flag algebra can be defined over any ring containing the rationals, this allows the evaluation of an arbitrary symbolic expression. Furthermore, a flag algebra can be defined for any combinatorial theory. Often in a richer structure (for example in graphs with colored vertices) additional extremal combinatorial tricks can be expressed (for example vertex extendability [CL24], or stability [BHLP16]), providing spectacular improvements for the computer generated proofs.

3.5 Satisfiability Coding Lemma Inductively

Switching lemma style arguments provide a way to reduce the depth of a small sized circuit or formula. Therefore, many constant depth circuit or formula lower bounds follow an inductive application of various restrictions or a switching lemma-based argument. Problems often require refined distributions on the restrictions [Hå86, Hå14, Bea90, Ros08, Ros18], but keep the inductive nature of the method, that the reduced circuit or formula is similar to the original one.

Regarding size lower bounds, for depth d circuits computing PAR, the switching lemma provides a

$$2^{\frac{1}{10}^{\frac{d}{d-1}} n^{\frac{1}{d-1}}}$$

bound. In contrast, the smallest known depth d circuit uses around

$$2^{n^{\frac{1}{d-1}}} n^{\frac{d-2}{2d-2}}$$

gates to compute PAR. This construction is described explicitly in Chapter 8. In particular, the construction provides

Theorem 21. *Given a factoring of n as $n = n_1 n_2 \ldots n_{d-1}$, there exists a circuit computing* PAR *on n input bits having depth d and layer $i < d$ has size $|V_i| = \frac{n}{n_1 n_2 \ldots n_i} 2^{n_i}$.*

Compared to the switching lemma, the satisfiability coding lemma [PPZ97] is only known to work for depth 3 circuits. However, it provides a much greater understanding of the behaviour of circuits and, as a consequence, a tighter circuit size lower bound for PAR. In particular, the coding lemma can provide a tight $n^{1/4} 2^{\sqrt{n}}$ size lower bound for depth 3 circuits computing PAR. The coding idea was used in various other settings [PPSZ05, Ama11, MST22, GPP+24], but no known generalization works in a similar inductive way as the switching lemma.

Chapter 8 attempts to address this challenge by introducing the notion of "cones", a generalization of critical clauses from [PPZ97]. Cones provide a combinatorial structure allowing the usage of a satisfiability coding lemma-style arguments in any function, thus offering a step towards extending the satisfiability coding lemma to higher depths. However, unfortunately cones in a given layer of a circuit are only partially related to cones in the next layer, causing the induction to fail in general. To bridge this gap, a new structural condition "nice" will be introduced, which informally states that the number of cones is the same as the number of satisfying assignments (or unsatisfying assignments, depending on $\wedge$ or $\vee$ gates). See Section 8.3.2 for the precise definition. This condition is satisfied by a broad class of circuits, including known optimal constructions, although examples of circuits exist that violate this structural condition, which will be further discussed in Chapter 8. Under this structural assumption, a corollary of the main result of the chapter Theorem 64 provides tight size lower bounds for PAR circuits.

Theorem 22. *Suppose a depth d, circuit $\mathcal{C}$ computes* PAR *on n bits, with each gate in $\mathcal{C}$ nice. Find a (not necessarily integer) sequence $n_1, n_2, ..., n_{d-1}$ such that $|V_i(\mathcal{C})| = \frac{n}{n_1 n_2 ... n_i} 2^{n_i - 1}$ for all $i \leq d$. If each $n_i \geq 2$, then it must hold that $\prod_i n_i \geq n$.*

Chapter 4

CLIQUE as an AND of Polynomial-Sized Monotone Constant-Depth Circuits

4.1 Introduction

The simplest circuit expressing CLIQUE is an OR of monomials, one for each possible clique. The top gate has fan-in $\binom{n}{k} \approx n^k$. The circuit complexity of CLIQUE_k is well studied. Under a constant-depth restriction, Lynch [Lyn86] established the first superpolynomial size lower-bound. This was subsequently refined in [Bea90] and finally in [Ros08], establishing an $\Omega\left(n^{k/4}\right)$ size lower-bound. Regarding monotone circuits, first Razborov [Raz85] proved a super polynomial size lower bound using the sunflower lemma from Erdős and Rado [ER60]. Further work improved this [AB87, CKR20] establishing the current best $n^{\Omega(k)}$ size lower-bound when $k \leq n^{1/3-o(1)}$.

Dually to the above, one can express CLIQUE as monotone AND of monomials, one for each maximal graph without k-clique. The number of maximal graphs without triangles was investigated in [BP14], where they show that their number is asymptotically $2^{n^2/8+o(n)}$. The same paper (in remark 7) constructs $2^{\frac{1-1/(k-1)+o(1)}{4}n^2}$ many maximal graphs without k-clique,

but the method only works for small $k < \log_{4/3}(n)$. Note that any maximal graph without k-clique can be extended to a maximal graph without k'-clique, by simply adding $k' - k$ new vertices, such that each new vertex has maximum degree. This gives a

$$\exp\left(\frac{\ln 2}{4}(1 - 1/\log_{4/3}(n-k) + o(1))(n-k)^2\right)$$

lower bound on the number of maximal k-clique free graphs, when $k > \log_{4/3}(n)$. Correspondingly, there is a monotone, depth-2 circuit, with top AND gate, computing k-CLIQUE, with top fan-in lower bounded by the same value.

The motivation for studying the type and fan-in of the top gate comes from descriptive complexity. In descriptive complexity, computational complexity classes are characterized by the power of logical languages needed to define problems over finite structures, like graphs [Imm98, Imm95]. First-order (FO) logic allows quantification over individual elements, like vertices in a graph, while second-order (SO) logic also allows quantification over sets or relations of elements like sets of edges. A second order logic is monadic if its second-order quantifiers range only over sets (unary relations). Instead of measuring the classical computational resources required by an algorithm to solve a problem, the logical resources needed to describe the problem itself are examined. Problems are viewed as properties of finite models, and the logics are studied, whether they are powerful enough to express these properties. Any second order sentence with only leading unary second order quantifiers, directly translates to circuits with a 2^{cn} fan-in top gate, followed by constant-depth polynomial-sized circuits.

A key insight and foundational result of descriptive complexity is Fagin's Theorem [Fag74], which demonstrated one of the first connections between logic and computation: the complexity class NP precisely corresponds to the set of properties expressible in second-order existential logic SO$\exists$ over finite structures. This provides a machine-independent characterization of NP. Further research extended this approach [Imm82, Imm87, Var82], showing that other major complexity classes also have natural logical counterparts. For instance

- P is captured by first-order logic augmented with a least fixed point operator FO(LFP),

- NSPACE[$\log n$] corresponds to first-order logic with a transitive closure operator FO(TC),

- PSPACE is captured by first-order logic with a partial fixed point operator FO(PFP).

Descriptive complexity has also yielded significant insights in other areas, such as database query language theory [CH82]. This line of research has settled a major, long-standing question in complexity theory by proving that NSPACE[$s(n)$] is closed under complementation for all $s(n)$ greater than or equal to $\log n$ [Imm88, Sze88].

Note that the trivial circuit construction calculating CLIQUE directly translates to a monotone monadic existential second order formula mMSO$\exists$. Showing that there is no universal second order sentence (SO$\forall$) for CLIQUE would separate NP and co-NP. A well-studied fragment of this theory is the collection of monadic second order sentences, where the second order quantifier is restricted to be unary. Graph connectivity famously separates monadic NP from monadic co-NP [FSV93]. It is an interesting open question if CLIQUE can be expressed in MSO$\forall$. This chapter progresses towards answering the natural question, finding the minimum number of monotone, constant-depth, and polynomial-sized circuits needed, such that their AND computes CLIQUE.

Theorem 13. *For any c_d, c_s constants, large enough n and $\log(n)^{c_d+6} < k$, one cannot have less than $2^{n/4k}$ many monotone circuits $\{f_j\}$ on $\binom{n}{2}$ input bits, each with depth and size bounded by c_d and n^{c_s} respectively, such that $\bigwedge_j f_j$ computes k-CLIQUE.*

Compared to classical circuit bounds for CLIQUE, this result is surprisingly strong, due to the restricted setting. This shows that OR gates are much better at expressing CLIQUE than AND gates in this sense. The result follows from the main lemma of this chapter, that monotone constant-depth polynomial-sized circuits with zero-error on YES instances have $2^{-n/4k}$ correlation with cliques on NO instances. This gives a strong asymmetric result; note that a single monomial, stating that a certain k subset is a clique, has zero error on NO instances, but on YES instances, has $\binom{n}{k}^{-1}$ correlation.

The main novelty of the proof is an approximation of a monotone CNF circuit having larger fan-in, by a monotone DNF circuit having a small fan-in, where this approximation is

correct on most of the relevant cliques. Approximation of Boolean functions with depth 2 circuits was initially investigated by [O'D07]. Continuing this line of work, [BHST14] showed strong one-sided bounds for all monotone functions.

Section 4.2 contains the main notation used in this chapter, Section 4.3 covers the main definitions, lemmas and an overall view of the proof. Section 4.4 contains all the technical steps and proofs of lemmas and theorems. The chapter finishes with concluding thoughts and open problems.

4.2 Conventions

For short write $K(X) = \binom{X}{2}$ for the complete graph on the vertex set X. Use A, B for subsets of $[n]$. Similarly, write G, H for subsets of $K([n])$, and identify them with graphs. k is a number that can depend on n, and use X, Y, Z for subsets of $\binom{[n]}{k}$. The curly versions represent families, so $\mathcal{A}, \mathcal{B}$ are used for families of subsets of $[n]$. Similarly $\mathcal{G}, \mathcal{H}$ are families of subsets of $K([n])$. For a family $\mathcal{A}$ write $\mathcal{K}(\mathcal{A}) = \{K(A) : A \in \mathcal{A}\}$.

Monotone circuits are represented using the letters f, g, and h. Given a circuit with input set U, for any $A \subseteq U$ write $f(A)$ for the value of the circuit on input A. Therefore they can be used as functions $f : \{0,1\}^U \to \{0,1\}$. For convenience use $f = g$ to mean that f and g compute the same function, not that the circuits are the same. The size $|f|$ is the number of nodes in the circuit. For f, g on the same input set U, write $f \leq g$ if $\forall A \subseteq U\ f(A) \leq g(A)$. For given $\mathcal{A} \subseteq \{0,1\}^U$ the DNF with monomials defined by $\mathcal{A}$ is $i_{\mathcal{A}} : \{0,1\}^U \to \{0,1\}$. This gives the function

$$i_{\mathcal{A}}(B) \mapsto \begin{cases} 1 & \text{if } \exists A \in \mathcal{A}\ (A \subseteq B) \\ 0 & \text{otherwise.} \end{cases}$$

The letters ρ, σ are monotone restrictions. On an input set U they are functions $\rho : U \to \{1, \star\}$. Given a restriction ρ and a circuit f the restricted circuit $f_\rho : \{0,1\}^{\rho^{-1}(\star)} \to \{0,1\}$ is the circuit where every input source from $\rho^{-1}(1)$ is set to constant true. Given an input set U, use R_U^p as a distribution on monotone restrictions on U, that maps $\rho(u)$ to $\star$ with probability

p (and therefore maps to 1 with probability $1-p$) independently for each $u \in U$.

4.3 Outline

The result follows from the following proposition on a single bounded depth and size monotone circuit.

Proposition 23. *For any c_d, c_s constants, large enough n and any monotone circuit f on $\binom{n}{2}$ input bits, with depth $d(f) \leq c_d$ and size $|f| \leq n^{c_s}$ such that $f \geq i_{\mathcal{K}\left(\binom{[n]}{k}\right)}$, f must also satisfy*

$$\underset{G \sim \mathrm{ER}(n,1-p)}{\mathbb{P}}(f(G)) \geq 1 - 2^{-n/3k}$$

where

$$p = O(1)\log(n)^{-c_d-4}.$$

The method is a switching of the bottom layers between small ($O(\log(n)^2)$) fan-in DNF and medium ($O(n/k)$) fan-in CNF. The rationale behind the numbers is the desire to ensure that the circuits remain ineffective at approximating CLIQUE. This holds true intuitively for CNFs even when larger fan-ins are allowed. However, for DNFs, a stronger fan-in restriction is necessary. To express this correlation with CLIQUE, the following approach will be employed:

Definition 24. *For a monotone restriction $\rho : K([n]) \to \{1, \star\}$ and a monotone circuit $f : \{0,1\}^{K([n])} \to \{0,1\}$ both on graphs, write $Z_\rho(f)$ for the maximal collection of cliques, where containing any in the input, forces the function to return true. It is the largest $Z_\rho(f) \subseteq \binom{[n]}{k}$ where it holds that $\left(i_{\mathcal{K}(Z_\rho(f))}\right)_\rho \leq f_\rho$.*

The main Lemma of the chapter shows that the CNF to DNF switching reduces the possible cliques forcing the circuit to true (the set $Z(f)$) by a negligible amount.

Lemma 25 (CNF to DNF switching with small clique error)**.** *For any monotone s-CNF $f : \{0,1\}^{K([n])} \to \{0,1\}$ and a monotone restriction $\rho : K([n]) \to \{1, \star\}$, one can find a*

monotone t-DNF g, that satisfies $g_\rho \le f_\rho$ *and*

$$\left|Z_\rho(f) \setminus Z_\rho(g)\right| \le \binom{n}{k}\left(s\frac{k}{n}\right)^{\sqrt{t/2}}.$$

The DNF to CNF switching requires a random restriction, but after the restriction the functions are the same

Lemma 26 (DNF to CNF switching)**.** *For any monotone t-DNF g, after taking* $\rho \sim R_U^{1/(2t)}$ *restriction, g can be written as an equivalent s-CNF with probability* $\ge 1 - 2^{-s-1}$.

The end result is a $O(\log(n)^2)$ fan-in DNF that is still true on a large number of cliques, therefore it holds that a random 0-1 assignment on the remaining variables satisfies the circuit with high probability. The probability that one of the DNF to CNF switching fails or that the final random assignment does not force the circuit to true, will be exponentially small, allowing the large fan-in bound on the output gate.

4.4 Proofs

The following is a list of simple observations, the proof is not included.

Observation 27.

1. *If* $\mathcal{G} \subseteq \mathcal{H}$ *then* $i_{\mathcal{G}} \le i_{\mathcal{H}}$.
2. *If* $f \le g$ *then* $Z(f) \subseteq Z(g)$.
3. *If* $\rho \sim R_U^p$ *and* $\sigma \sim R^q_{\rho^{-1}(\star)}$ *then* $\sigma \circ \rho \sim R_U^{pq}$.
4. *If* $G = \rho^{-1}(1)$ *where* $\rho \sim R^p_{K([n])}$ *then* $G \sim \mathrm{ER}(n, 1-p)$.

The idea for the proof of Lemma 25 is to build a monotone decision tree both by cliques and by edges simultaneously based on the formula. This gives enough control over the cliques they imply. The extension of edge sets to cliques can only decrease the function but the clique

implications $Z(f)$ will not change. The tree is pruned to include only clauses below the cut. This can be achieved with a minimal loss of cliques.

Proof of Lemma 25. First simplify the function f to only include edges from $\rho^{-1}(\star)$. If any of the clauses become trivial, then $f_\rho \equiv 1$ therefore $g \equiv 1$ works. Name the clauses $f_\rho = \bigwedge_{j=1}^{m} \left(\bigvee_{e \in q_j} e \right)$ where $|q_j| \leq s$.

Create two trees $T \subseteq T'$ from the following data.

1. T has its nodes labeled by subsets of $[n]$, while T' has the labels from subsets of $K([n])$.

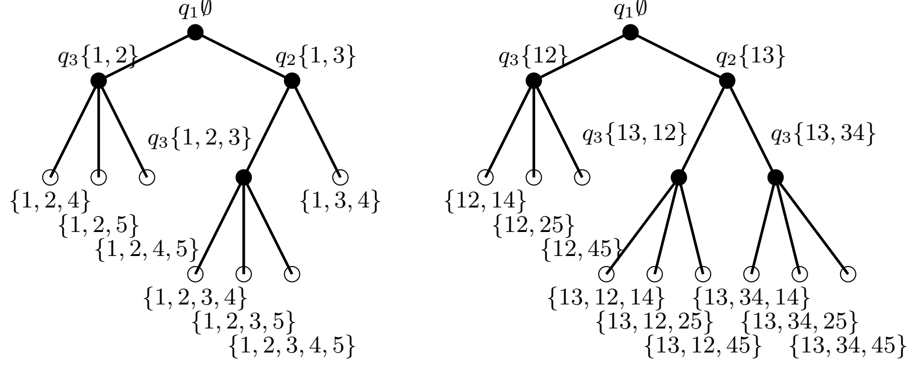
2. Every non-leaf has a clause associated from f_ρ both in T and T'.

3. For $v \in T$ denote $A(v) \subseteq [n]$ the label and $q_T(v)$ the associated clause. Similarly denote $G(w) \subseteq K([n])$ the label and $q_{T'}(w)$ for $w \in T'$ the clause.

Constructing T: The root $r \in T$ has $\emptyset = A(r)$. A node $v \in T$ is a leaf if $f_\rho(K(A(v))) = 1$. If $v \in T$ is not a leaf, find the clauses in f_ρ not satisfied by $K(A(v))$. Let the first clause be q_j, then assign $q_T(v) = q_j$. For $v \in T$ with a clause label $q_T(v)$ and for each edge $e \in q_T(v)$ not already satisfied by $K(A(v))$, add v' a child of v in T with label $A(v') = A(v) \cup e$.

Constructing T': The construction will create $T \subseteq T'$ an extended tree. It will also preserve $\bigcup G(v) = A(v)$. Accordingly, it holds that for $r \in T'$ root it holds that $\emptyset = G(r)$. A node $w \in T'$ is a leaf if $f_\rho(G(w)) = 1$. If $v \in T'$ is not a leaf, find the clauses in f_ρ not satisfied by $G(v)$. As $G(v) \subseteq K(A(v))$, if $v \in T$ is not a leaf, the same clause selected for $v \in T$ is also unsatisfied in $G(v)$. Assign the same $q_T(v) = q_{T'}(v) = q_j$. If $v \in T$ is a leaf, choose any unsatisfied q_j. For $v \in T'$ with a clause label $q_{T'}(v)$ and for each edge $e \in q_{T'}(v)$ not already satisfied by $G(v)$, add v' a child of v in T' with label $G(v') = G(v) + e$. Note that this can be performed such that the same v also satisfies $A(v') = A(v) \cup e$.

Call a node's depth its distance from root. Let $\mathcal{A}_d$ be the collection of $A(v)$ where v is a leaf with depth $\leq d$ in T. Let $\mathcal{B}_d$ be the collection of $A(v)$ where v is any node in T with depth exactly d (not necessarily a leaf). Similarly define $\mathcal{G}_d$ to be the collection of $G(w)$ for $w \in T'$ leaf node of depth $\leq d$. Let $d(T), d(T')$ be the maximal depth of the trees.



The trees on formula $f = q_1 \wedge q_2 \wedge q_3 = (12 \vee 13) \wedge (12 \vee 34) \wedge (14 \vee 25 \vee 45)$ with clauses numbered accordingly.

Claim 28 (Relations involving the $\mathcal{G}_d, \mathcal{A}_d, \mathcal{B}_d$ sets)**.**

1. $i_{\mathcal{G}_{d(T')}} = f_\rho$

2. $i_{\mathcal{K}(\mathcal{A}_{d(T)})} \leq f_\rho$

3. $Z_\rho\left(i_{\mathcal{K}(\mathcal{A}_{d(T)})}\right) = Z_\rho(f)$

4. $i_{\mathcal{K}\left(\mathcal{A}_{d(T)}\right)} \leq i_{\mathcal{K}\left(\mathcal{A}_d \cup \mathcal{B}_{d+1}\right)}$

5. $|\mathcal{B}_d| \leq s^d$

Proof of Claim 28.

1. Working in $U = \rho^{-1}(\star)$, for any $u \subseteq U$ it holds that $f_\rho(u)$ iff for all q_j it is true that $u \cap q_j \neq \emptyset$. Therefore every non leaf $w \in T'$ has an edge $e_w \in q_{T'}(w)$ such that $e_w \in u$. Following the label set increments $G(w)$ to $G(w) \cup \{e_w\}$, provides a path from root to a leaf w such that each label in that path is a subset of u resulting in $G(w) \subseteq u$ and therefore $i_{\mathcal{G}_{d(T')}}(u) = 1$. This gives $i_{\mathcal{G}_{d(T')}} \geq f_\rho$. For the other direction take u such that $i_{\mathcal{G}_{d(T')}}(u) = 1$. Then there is a leaf $w \in T$ where $G(w) \subseteq u$. By definition of a leaf, $G(w)$ already satisfies f_ρ giving that by monotonicity $f_\rho(u) = 1$.

2. Consider any $u \subseteq U$ such that $i_{\mathcal{K}\left(\mathcal{A}_{d(T)}\right)}(u) = 1$, then there must be a leaf $v \in T$ and a corresponding $A(v) \in \mathcal{A}_{d(T)}$ with $K(A(v)) \subseteq u$. Then as v is a leaf $K(A(v))$ satisfies f_ρ and by monotonicity so does u.

3. Using point 2 in Observation 27 and the previous point, it holds that $Z_\rho\left(i_{\mathcal{K}\left(\mathcal{A}_{d(T)}\right)}\right) \subseteq Z_\rho(f)$. For the other direction, take any $B \in Z_\rho(f)$. Then $K(B)$ satisfies f_ρ which by above agrees with $i_{\mathcal{G}_{d(T')}}$. Take a leaf $w \in T'$ where $G(w) \subseteq K(B)$. Then as $T \subseteq T'$ the path from the root to w must contain a leaf from T, say it is v. Then $G(v) \subseteq G(w) \subseteq K(B)$ therefore $A(v) = \bigcup G(v) \subseteq \bigcup G(w) \subseteq B$ so $K(B)$ satisfies $i_{\mathcal{K}\left(\mathcal{A}_{d(T)}\right)}$ as required.

4. Suppose for $u \subseteq U$ it also holds that $i_{\mathcal{K}(\mathcal{A}_{\lceil(T)})}(u) = 1$, then there must be a leaf $v \in T$ and a corresponding $A(v) \in \mathcal{A}_{d(T)}$ with $K(A(v)) \subseteq u$. If the depth of v is $\leq d$ then $v \in \mathcal{A}_d$ and $i_{\mathcal{K}(\mathcal{A}_d)}(u) = 1$. Otherwise there must be a path from root to v going through a node at depth $d+1$, giving that $i_{\mathcal{K}(\mathcal{B}_{d+1})}(u) = 1$ proving the claim.

5. Every set in $\mathcal{B}_d$ can be identified with a path from the root to a node at level d. As the tree branches at each step to at most s new nodes, the mentioned bound follows.

□

Combining the above with point 2 of Observation 27 and by noting for all d it holds that $\mathcal{A}_d \subseteq \mathcal{A}_{d(T)}$,

$$Z_\rho\left(i_{\mathcal{K}(\mathcal{A}_d)}\right) \subseteq Z_\rho(f) \subseteq Z_\rho\left(i_{\mathcal{K}\left(\mathcal{A}_d \cup \mathcal{B}_{d+1}\right)}\right)$$

this gives

$$\left|Z_\rho(f) \setminus Z_\rho\left(i_{\mathcal{K}(\mathcal{A}_d)}\right)\right| \leq \left|Z_\rho\left(i_{\mathcal{K}\left(\mathcal{A}_d \cup \mathcal{B}_{d+1}\right)}\right) \setminus Z_\rho\left(i_{\mathcal{K}(\mathcal{A}_d)}\right)\right| \leq \sum_{B \in \mathcal{B}_{d+1}} \binom{n-|B|}{k-|B|}.$$

Note that every $B \in \mathcal{B}_{d+1}$ is constructed in $d+1$ steps, each increasing the cardinality by at

least 1 but at most 2, therefore $d+1 \leq |B| \leq 2d+2$ meaning

$$\left| Z_\rho(f) \setminus Z_\rho\left(i_{\mathcal{K}(\mathcal{A}_d)}\right)\right| \leq s^{d+1}\binom{n-(d+1)}{k-(d+1)} \leq \binom{n}{k}\left(s\frac{k}{n}\right)^{d+1}$$

as in $i_{\mathcal{K}(\mathcal{A}_d)}$ each clause has at most $\binom{2d}{2}$ edges, setting d such that $t = \binom{2d}{2}$ gives the result with $g = i_{\mathcal{K}(\mathcal{A}_d)}$. □

Proof of Lemma 26. Build a tree T' the same way as in Lemma 26 (without caring about any T) using the clauses of g. Write $\mathcal{H}_d$ for the collection of $G(w)$ sets where w is a node at depth exactly d. Using this tree, write g_ρ as a s-CNF if all the $\mathcal{H}_{s+1}$ OR clauses are satisfied by ρ. Similar to point 5 in Claim 28, there is $|\mathcal{H}_{s+1}| \leq t^{s+1}$ and each $H \in \mathcal{H}_{s+1}$ has exactly $s+1$ edges. Therefore a $\rho \sim R_U^{1/(2t)}$ satisfies all the OR clauses in $\mathcal{H}_{s+1}$ with probability lower bounded by

$$1 - |\mathcal{H}_{s+1}|\left(\frac{1}{2t}\right)^{s+1} \geq 1 - 2^{-s-1}$$

as needed. □

Recall Proposition 23.

Proposition 23. *For any c_d, c_s constants, large enough n and any monotone circuit f on $\binom{n}{2}$ input bits, with depth $d(f) \leq c_d$ and size $|f| \leq n^{c_s}$ such that $f \geq i_{\mathcal{K}\left(\binom{[n]}{k}\right)}$, f must also satisfy*

$$\underset{G\sim\mathrm{ER}(n,1-p)}{\mathbb{P}}(f(G)) \geq 1 - 2^{-n/3k}$$

where

$$p = O(1)\log(n)^{-c_d-4}.$$

Proof of Proposition 23. The Proposition follows from the combination of the two switchings.

First transform the circuit to having alternating AND and OR layers of gates and extend the bottom with dummy gates to have fanin 1. This results in f' agreeing with f on all inputs, while still $|f'| \leq n^{c_s+1}$ and $d(f') \leq c_d + 1$. Fix $t = 2(c_s+2)^2\log(n)^2$ and $s = n/2k$;

this is to ensure that $\left(s\frac{k}{n}\right)^{\sqrt{t/2}} = n^{-(c_s+2)}$ The proof relies on the changing of the bottom layer between s-CNFs and t-DNFs, after each switch, merging with the layer above therefore reducing the depth.

Provided the bottom layer is a t-DNF, use Lemma 26 for each DNF, introducing at depth i a random $\rho_j \sim R^{1/(2t)}_{\rho^{-1}_{(<i)}(\star)}$ where write $\rho_{(<i)} = \underset{j<i}{\circ}\rho_j$ for the composition of previous restrictions. The change is unsuccessful with probability $\leq 2^{-s-1}$ at each DNF, otherwise results in a s-CNF.

Provided the bottom layer is a s-CNF use Lemma 26. Each gate g in the bottom CNF layer is replaced with g' a t-DNF satisfying $g' \leq g$ and by the selection of parameters, it holds that

$$\left|Z_{\rho_{(\leq i)}}(g) \setminus Z_{\rho_{(\leq i)}}(g')\right| \leq \binom{n}{k}\left(s\frac{k}{n}\right)^{\sqrt{t/2}} = \binom{n}{k}n^{-c_s-2}.$$

The probability that any of the DNF to CNF switching fails is $\leq n^{c_s+1}2^{-s-1}$ by the union bound. Write $\rho = \rho_{(<c_d+2)}$. The total number of cliques lost from CNF to DNF switches is at most $\binom{n}{k}n^{-1}$ from the union bound. This gives that with probability $\geq 1 - n^{c_s+1}2^{-s-1}$ it is possible to find f'' a t-DNF such that $(f'')_\rho \leq (f')_\rho$ and by noting $Z(f') = \binom{[n]}{k}$ it holds that $\left|Z_\rho(f'')\right| \geq \binom{n}{k}(1 - n^{-1})$.

Suppose $f'' = \bigvee_j \left(\bigwedge_{e\in q_j} e\right)$. To conclude the proof disjoint sets among q_j will be selected. Iteratively pick the sets r_j from q_j such that they are disjoint. Start with $r_1 = q_1$ and suppose inductively that $r_1, ..., r_x$ are selected so far and they are all disjoint. Consider

$$X = \{A \in Z_\rho(f'') : \forall j \leq x,\ r_j \cap K(A) = \emptyset\}$$

and notice that

$$|X| \geq \binom{n}{k}\left(1 - n^{-1} - xs\frac{k(k-1)}{n(n-1)}\right)$$

since each $|r_j| \leq s$ and the possible cliques containing one particular edge has size $\binom{n-2}{k-2}$. Therefore each edge in r_j removes this number of cliques at most. Provided $|X| > 0$ it is possible to pick $A \in X$ and as $f''(K(A) \cup \rho^{-1}(1)) = 1$ there must be one q_j satisfied by the

inclusion of $K(A)$ giving that q_j is disjoint from $\{r_j : j \leq x\}$ and it is possible to extend the collection. Therefore, at least $x = \frac{n(n-1)}{2sk(k-1)}$ many disjoint clauses can be constructed, giving that after assigning each remaining $\rho^{-1}(\star)$ edges to $\star$ with probability $1/2t$ giving an extra $\sigma \sim R^{1/2t}_{\rho^{-1}(\star)}$ restriction, it holds that

$$\mathbb{P}\left(f''\left(\sigma^{-1}(1)\right)\right) \geq 1 - \prod_{j=1}^{x}\left(1 - \left(1 - \frac{1}{t}\right)^{|r_j|}\right) \geq 1 - \left(1 - \frac{1}{e^2}\right)^{\frac{n(n-1)}{2\log(n)k(k-1)}}.$$

This combined with the probability that any of the c_d many switching fails, gives the bound

$$\mathbb{P}\left(f\left((\sigma \circ \rho)^{-1}(1)\right)\right) \geq 1 - n^{c_s+1}2^{-\frac{n}{2k}-1} - \left(1 - \frac{1}{e}\right)^{\frac{n(n-1)}{2\log(n)k(k-1)}} \geq 1 - 2^{-\frac{n}{3k}}.$$

At each restriction a $\sigma, \rho_j \sim R^{1/(2t)}_U$ is used. Every second layer uses one such restriction, and an additional restriction used at the last step, giving a total $\sigma \circ \rho \sim R^p_{K([n])}$ where

$$p \geq \left(\frac{1}{2t}\right)^{c_d/2+2} = (2c_s + 2)^{-c_d-4}\log(n)^{-c_d-4}.$$

□

Proof of Theorem 13. Use Proposition 23 with the same c_d and c_s constants. This gives every AC^0 circuit satisfying the constraints is true on a $G \sim R^p_{K([n])}$ input with probability $1 - 2^{-n/3k}$, therefore all the f_j is true on input G with probability $1 - o(1)$ using the union bound, but note that for $\log(n)^{c_d+6} < k$ the probability that a clique appears in G is upper bounded by

$$\binom{n}{k}(1-p)^{\binom{k}{2}} \leq 2^{\log(n)k - \frac{k^2}{c'(c_d,c_s)\log(n)^{c_d+3}}} = o(1)$$

using p from Proposition 23. Therefore the circuit cannot express clique as there is a nonzero probability that G forces all the f_j to true, but G still has no clique. □

4.5 Concluding Remarks

As outlined in the introduction, one can write CLIQUE as an OR of $\binom{n}{k}$ many monotone polynomial-sized circuits, with a top OR gate. This gives a $\binom{n}{k}^{-1}$ one-sided correlation between monotone polynomial-sized circuits and clique, which is correct on all negative instances. One can ask the dual question:

Question 29. *What is $S(n,k)$, the maximum one-sided correlation between monotone poly-sized circuits and k-CLIQUE on n vertices, which is correct on all positive instances?*

Here, a partial answer is given, that $S(n,k) \leq 2^{-n/4k}$ (for large enough k). There remains a considerable gap between the best-known constructions and this bound, it would be interesting to know tighter bounds for $S(n,k)$.

Conjecture 30. *$S(n,k) = 2^{-\omega(n)}$ for k in a suitable range (for example $k = n^{\alpha}$ for some $0 < \alpha < 1$).*

This would also imply that monotone monadic second-order sentences with universal monadic predicates are not strong enough to express k-CLIQUE, while monotone existential second-order sentences can express it. Looking at the proof of Lemma 25, the inequality

$$\sum_{B \in \mathcal{B}_{d+1}} \binom{n-|B|}{k-|B|} \leq s^{d+1} \binom{n-(d+1)}{k-(d+1)}$$

is used, following from $d+1 \leq |B| \leq 2d+2$. A better approximation of the set sizes on average could provide a stronger result, in particular if all $|B| \approx 2d+2$, then the resulting bound is $S(n,k) \leq 2^{-O(n^2/k)}$. The following stronger conjecture captures this:

Conjecture 31. *It holds that $S(n,k) \leq 2^{-\Theta(n^2/k)}$ for k in a suitable range.*

Note that while this chapter discusses the monotone question, as the method heavily uses that condition, the general question is equally interesting:

Question 32. *What is the maximum one-sided correlation between polynomial-sized circuits and k-CLIQUE on n vertices, which is correct on all positive instances?*

Chapter 5

Depth 3 Circuits Computing THR

5.1 Introduction

In this chapter, depth 3 circuits computing the THR_r function with a top AND gate and k-bounded bottom fan-in will be considered. Due to the restricted bottom fan-in setting, size bounds for the middle layer will be derived for circuit computing the THR_r function.

Bounding the bottom fan-in by k is not an unusual assumption. This can be guaranteed for any circuit with small enough bottom layer, after a random (or greedy) restriction [HJP95, Hå86]. Additionally, for small k, stronger lower bounds are known, even for circuits. Paturi, Pudlák and Zane showed that the minimum size of a depth 3 circuit computing the parity function on n variables is at least $2^{n/k}$, for any $k = O(\sqrt{n})$ [PPZ97]. It is conjectured that any circuit with bottom fan-in bounded by k, computing MAJ, requires $k^{\Omega(n/k)} = 2^{\Omega(n \log k/k)}$ size for the middle layer, matching the upper bound obtained by a natural divide and conquer construction.

Compared to other literature, the assumption in this chapter, that the top gate is an AND, is an unusual formulation. That is in order to make the connection between hypergraph properties easier. In the dual case, the middle layer consists of k-CNFs, making it historically the preferred setting. Note, in THR_r logical duality cannot be ignored (unlike PAR or MAJ). Switching the AND and OR gates, the resulting circuit computes $\neg\text{THR}_r(\neg A) = \text{THR}_{n-r}(A)$. As THR_r is a monotone function, this chapter additionally explores the question restricted

to monotone depth 3 and bounded bottom fan-in circuits.

5.1.1 Overview of the Results

The most efficient known method for computing the THR_r function, in terms of the size of the middle layer, employs a block construction approach. This construction will be made precise in Section 5.2, relating the size of the middle layer to the number of decompositions $T_{\underline{b},r}$. Recall the definition of decompositions.

Definition 14. *For a sequence* $\underline{b} = (b_0, b_1, ..., b_{l-1})$ *and* $0 \le r \le \sum_{j\in[l]} b_j$*, write*

$$T_{\underline{b},r} = \left\{ (t_0, t_1, ..., t_{l-1}) \ : \ 0 \le t_j \le b_j, \ \sum_{j\in[l]} t_j = r \right\}$$

for the set of decompositions.

The quantity $\left|T_{\underline{b},r}\right|$ is exactly the decompositions of the number r into l many terms, each bounded by the corresponding b_j value.

Theorem 15. *Suppose the* $\underline{B} = (B_0, B_1, ..., B_{l-1})$ *sets partition the input bits* $[n]$*, with corresponding sizes* $\underline{b} = \big(|B_0|, \ |B_1|, \ ..., \ |B_{l-1}|\big)$*. Then there is a* $m\Pi_3^k$*-formula, with* $k = \max(\underline{b})$ *and* $\left|T_{\underline{b},r}\right|$ *gates in the middle layer, computing* THR_r*.*

Additionally, any circuit that follows a block structure must have size of the middle layer greater than or equal to the above construction.

Theorem 16. *Suppose the* $\underline{B} = (B_0, B_1, ..., B_{l-1})$ *sets partition the input bits* $[n]$*, with corresponding sizes* $\underline{b} = \big(|B_0|, \ |B_1|, \ ..., \ |B_{l-1}|\big)$*. If a* Π_3^k*-circuit*

$$\bigwedge_{i\in[m]} \bigvee_{C\in D_i} \bigwedge_{l\in C} l$$

computes THR_r*; where for each term* $C \in D_i$ *there is some block* B_j*, such that* C *only contains literals from* B_j*; then the size of the middle layer is at least* $m \ge \left|T_{\underline{b},r}\right|$*.*

Section 5.3 connects the unconditional question with various extremal hypergraph problems. A simple connection is established that relates the one-sided correlation between Π_3^{k-} and THR_r, with a question in extremal combinatorics. This correspondence is based on k-limits, following the work of Håstad, Jukna and Pudlák [HJP95]. The section also proves a similar connection between the monotone variant of the problem and the generalized hypergraph Turán question.

Finally, results for the extremal combinatorial questions are showcased in Section 5.3.1. The case with $k = 2$ is fully resolved. The bound from [HJP95] is slightly improved using simple combinatorial ideas. Chapter 6 establishes bounds on the generalized hypergraph Turán problem. The corollaries of that chapter are interpreted and compared with other bounds. In addition, several cases where the monotone and non-monotone question agree are highlighted.

5.2 Block Restriction

For THR_r, the best known construction uses a divide and conquer idea. The input bits are divided into blocks, of size $\underline{b} = (b_0, b_1, ..., b_{l-1})$. Block j can encode the number of 1s inside with $\log(b_j)$ bits, therefore a large circuit with size of middle layer

$$2^{\sum_{b_j \in \underline{b}} \log(b_j)} = \prod_{b_j \in \underline{b}} b_j$$

can compute any threshold function.

This section shows how the block construction can be made more precise, to include information about r, and that it can be made monotone. Under the assumption that the clauses can only include bits from one block only, the section also shows that the following construction is optimal (in terms of minimal size for the middle layer).

Theorem 15. *Suppose the* $\underline{B} = (B_0, B_1, ..., B_{l-1})$ *sets partition the input bits* $[n]$*, with corresponding sizes* $\underline{b} = \left(|B_0|,\ |B_1|,\ ...,\ |B_{l-1}|\right)$*. Then there is a* $m\Pi_3^k$*-formula, with* $k = \max(\underline{b})$ *and with size of the middle layer equal* $\left|T_{\underline{b},r}\right|$*, computing* THR_r*.*

Proof of Theorem 15. For $\underline{t} \in T_{\underline{b},r}$ write

$$D_{\underline{t}} = \bigcup_{j \in [l]} \binom{B_j}{t_j + 1}.$$

Then the formula is simply

$$\mathrm{THR}_r = \bigwedge_{\underline{t} \in T_{\underline{b},r}} \bigvee_{C \in D_{\underline{t}}} \bigwedge_{x \in C} x. \tag{5.1}$$

The formula in the right hand side of Equation (5.1) is monotone, as required. Each C is an element of some $\binom{B_j}{t}$, therefore, has at most $\max_{j \in [l]} |B_j|$ size. Clearly the formula has $\left|T_{\underline{b},r}\right|$ gates in the middle layer. Due to monotonicity, to show the formula computes the threshold function, it is enough to only consider r and $r+1$ sized inputs.

For any r-sized input $A \in \binom{[n]}{r}$, consider

$$\underline{t} = \big(\ |B_0 \cap A|, |B_1 \cap A|, ..., |B_{l-1} \cap X|\ \big) \in T_{\underline{b},r}.$$

The DNF

$$\bigvee_{C \in D_{\underline{t}}} \bigwedge_{x \in C} x$$

is false, since from each block B_j, at least $t_j + 1$ elements are required to satisfy any of the clauses in that block, while A only has t_j only. This forces the entire expression to be false.

On the other hand, given $A \in \binom{[n]}{r+1}$ and any $\underline{t} \in T_{\underline{b},r}$, there must be a block B_j where $|B_j \cap A| > t_j$ by the pigeon hole principle using $\sum_{j \in [l]} t_j = r$ and $|A| = r + 1$. This means there is some $C \subseteq B_j \cap A$ clause from $C \in \binom{B_j}{t_j+1} \subseteq D_t$, which is satisfied, therefore the entire formula is true. This shows equality in Equation (5.1). □

An important property of this construction is that the input is partitioned into blocks and the clauses only contain terms from exactly one block. Under this assumption, next it is shown that $\left|T_{\underline{b},r}\right|$ gates in the middle layer is the minimum possible.

Theorem 16. *Suppose the $\underline{B} = (B_0, B_1, ..., B_{l-1})$ sets partition the input bits $[n]$, with cor-*

responding sizes $\underline{b} = \left(|B_0|,\ |B_1|,\ \ldots,\ |B_{l-1}|\right)$*. If a* Π_3^k*-circuit*

$$\bigwedge_{i\in[m]} \bigvee_{C\in D_i} \bigwedge_{l\in C} l$$

computes THR_r*; where for each term* $C \in D_i$ *there is some block* B_j*, such that* C *only contains literals from* B_j*; then the size of the middle layer is at least* $m \geq \left|T_{\underline{b},r}\right|$*.*

Proof of Theorem 16. Suppose

$$\mathrm{THR}_r = \bigwedge_{i\in[m]} \bigvee_{C\in D_i} \bigwedge_{x\in C} x,$$

with the C sets only containing literals from one block, and $m < \left|T_{\underline{b},r}\right|$.

As an abuse of notation, write D_i for the sub-circuit that it computes, so

$$D_i(A) = \bigvee_{C\in D_i} \bigwedge_{x\in C} x(A).$$

Note each D_i has $D_i(A) = 1$ on each $A \in \binom{[n]}{r+1}$, otherwise the entire circuit would be incorrect on that input. On the other hand, write

$$T_i = \left\{ (|A \cap B_0|, |A \cap B_1|, ..., |A \cap B_{l-1}|) : A \in \binom{[n]}{r}, D_i(A) = 0 \right\}$$

to be the set that captures the block size patterns that can appear among r sized inputs, where D_i is correctly false.

All $A \in \binom{[n]}{r}$ has $\mathrm{THR}_r(A) = 0$, so there must be some D_i where $D_i(A) = 0$. This makes all block size patterns appear in the T_i sets, so

$$\bigcup_{i\in[m]} T_i = T_{\underline{b},r}.$$

The rest of the proof will show that $|T_i| \leq 1$ for each i, which implies the claim $m \geq |T_{\underline{b},r}|$.

Suppose by contradiction, there is an i where T_i has two distinct elements $\underline{t}^0$ and $\underline{t}^1$. By

definition, there are corresponding inputs A^0 and A^1 such that $D_i\left(A^0\right) = D_i\left(A^1\right) = 0$. Define the input $A^{\max}$ block-wise,

$$A^{\max} \cap B_j = \begin{cases} A^0 \cap B_j & \text{if } t_j^0 > t_j^1 \\ A^1 \cap B_j & \text{otherwise.} \end{cases}$$

In words, $A^{\max}$ chooses the larger sized part from each block between A^0 and A^1.

Suppose a given $C \in D_i$ is a subset of the block B_j. Both A^0 and A^1 are false on C, and since $A^{\max}$ agrees with one of them on the entirety of B_j, $A^{\max}$ must be false as well on C clause. This makes $D_i\left(A^{\max}\right) = 0$, a false input to the DNF D_i. $\underline{t}^0$ and $\underline{t}^1$ both sum to r, while being different, means that taking the larger value from each, results in a sequence that sums to a larger value. This makes $\left|A^{\max}\right| > r$, so D_i is false on an input with at least $r+1$ ones, and therefore the entire circuit false on that input, contradicting that the circuit agrees with THR_r. □

5.2.1 Size of $T_{\underline{b},r}$

The value of $\left|T_{\underline{b},r}\right|$ is hard to find in general. One can obtain the bound

$$\left|T_{\underline{b},r}\right| \leq \prod_{i\in[l]} b_i,$$

by ignoring the $\sum_{i\in[l]} t_i = r$ condition. By ignoring the $t_i \leq b_i$ condition, using the stars and bars argument (counting the location of the partition boundaries) one can obtain the bound

$$\left|T_{\underline{b},r}\right| \leq \binom{l+r-1}{r}$$

. Note that $\left|T_{\underline{b},r}\right| = \left|T_{\underline{b},n-r}\right|$, since one can replace a $(t_0, t_1, ..., t_{l-1})$ sequence with $(b_0 - t_0, ..., b_{l-1} - t_{l-1})$ to form a bijection between the two sets. This means, using the previous bound, that

$$\left|T_{\underline{b},r}\right| \leq \binom{l+n-r-1}{n-r}$$

holds too. The case where all $b_i = k$ is briefly discussed in [HM10], with the notation from there $\left|T_{(k,k,\ldots,k),r}\right| = |C_{[k]}(r, n/k)|$, without asymptotic results. The asymptotic results usually assume a fixed k, and a relation between n and r. In particular [Rat08] discusses the growth of $\left|T_{(k,k,\ldots,k),r}\right|$ around the center, when $n/2 = r + c\sqrt{r}$ for some absolute constant c. They derive the estimate

$$\left|T_{(k,k,\ldots,k),n/2-c\sqrt{n}}\right| \approx (k+1)^{n/k} \exp\left(\frac{-6c^2}{2+k}\right) \frac{1}{\sqrt{kn}} \Theta(1).$$

In the wider range where $r = cn$ for some $c \in [0, 1]$ constant, [Mal] shows that

$$\left|T_{(k,k,\ldots,k),cn}\right| \approx \left(c^{-c}(1-c)^{-(1-c)}\frac{k+1}{2}\right)^{n/k}.$$

Going back to the hardest instance, MAJ (where $c = 1/2$) the best known construction, based on Theorem 15 appears with b_i is as close to each other as possible and each with size around $k = \sqrt{n\log(n)/2}$. This provides a formula with size of the middle layer approximately $2^{(1+o(1))\sqrt{n\log n/2}}$. With the (strong) assumptions of Theorem 16 that the AND gates on the bottom layer can only include variables from at most one block, with all blocks having size $|B_i| = k$, the mentioned construction is optimal.

Note that the bound in Theorem 16 gets weaker as the block sizes increase. In fact the most general $\underline{b} = (n)$ gives a trivial n size lower bound on the number of gates required in the middle layer. The following section shows that the middle layer must be large without any condition on the bottom layer, by establishing a connection between this and a problem in extremal hypergraph theory.

5.3 Associated Hypergraph Problems

As noted in [FGT22], questions about the size of the middle layer of depth 3 circuits can be formulated as extremal problems in combinatorics. For the threshold function, the corresponding notion is captured by the following hypergraph property.

Definition 33. *For $k \leq g < r$, a g-graph G has property $P_{k,r}^{(g)}$, if for every $X \subseteq V(G)$, with $r < |X|$, there is a $C(X) \in \binom{X}{k}$ such that $\forall A \in G$ $(A \subseteq X \Rightarrow C(X) \nsubseteq A)$.*

Informally, $P_{k,r}^{(g)}$ means every X subset of points, larger than r, has some k sized core $C(X)$ that is not covered by the edges fully inside X. Note that $C(X)$ might be covered by all the edges of G, the definition requires the covering edges to also lie inside X. Notice $P_{k,r}^{(g)}$ is a hereditary hypergraph property. This hypergraph property was first defined, using a different language, in [HJP95]. To expand on the connection, if one considers $G \subseteq \{0,1\}^n$ as a set of vectors, the property means there is no upper-k-limit among vectors with weight greater than r (upper-k-limit, due to the setting with top AND gate). Using the notation of [HJP95], the negation of the property, for a given fixed X, would mean that for all $C \subseteq X$ with size k, one can find a vector $E \in G$, such that $E \leq X$ and $E|_C = X|_C$, which agrees with the definition.

The correspondence between the above hypergraph property and small circuits calculation the threshold function is captured by the following.

Theorem 34. *Given a k^--DNF f, write $G = \{A : f(A) = 0, |A| = g\}$ for the hypergraph formed by the g size unsatisfying assignments of f. If f is true on all assignments with size at least $r+1$, then G satisfies $P_{k,r}^{(g)}$. Conversely, if G is any r-graph satisfying $P_{k,r}^{(g)}$, then there is a k^--DNF whose g sized unsatisfying assignments are G, and is true on all assignments with size at least $r+1$.*

Proof of Theorem 34. Suppose G does not satisfy $P_{k,r}^{(g)}$, then there must exist some X with $|X| > r$ such that all $C \subset X$ with $|C| = k$ there is an $A \in G$ with $A \subset X$ and $C \subseteq A$. Write $f = \bigvee_i D_i$ and pick any k^--clause D_i. D_i has at most k many positive literals, write C_i to be that collection. If C_i is not a subset of X, then clearly $D_i(X) = 0$. Otherwise the property $P_{k,r}^{(g)}$ guarantees that there is an $A \in G$ unsatisfying assignment of f (and therefore of D_i) such that $C \subseteq A \subset X$. Since A is unsatisfying, it violates one literal from D_i, it can't violate a positive literal, as $C \subseteq A$, therefore it violates a negative literal from D_i. As $A \subseteq X$, the same negative literal is violated by X too, making $D_i(X) = 0$. As D_i was arbitrary, the entire DNF must be false on X, therefore f can't be true on all inputs with at least $r+1$ ones.

For the other direction take the DNF

$$f = \bigvee_{X \subseteq [n], |X| > r} \left(\bigwedge_{b \notin X} \neg b \bigwedge_{c \in C(X)} c \right),$$

where $C(X)$ is from Definition 33. Since $|C(X)| = k$, the resulting formula is a k^{-}-DNF. Note that the clause $D_X = \bigwedge_{b \notin X} \neg b \bigwedge_{c \in C(X)} c$ is true on the input X. On the other hand, take any $A \in G$ and any clause $D_X = \bigwedge_{b \notin X} \neg b \bigwedge_{c \in C(X)} c$. If $A \nsubseteq X$ then $\bigwedge_{b \notin X} \neg b$ is not satisfied. Otherwise $\bigwedge_{c \in C(X)} c$ is not satisfied as $C(X) \nsubseteq A$, making $D_X(A) = 0$ and A an unsatisfying assignment. □

Note that the maximum number of g sized unsatisfying assignments that can appear in a k^{-}-DNF f, such that f is true on all inputs with at least $r+1$ ones, is by definition $\pi\left(n; P_{k,r}^{(g)}\right)\binom{n}{g}$. Therefore k^{-}-DNFs can have at most $\pi\left(n; P_{k,r}^{(r)}\right)$ one-sided correlation compared to THR_r. Any upper-bound on $\pi\left(n; P_{k,r}^{(r)}\right) \leq \alpha$ translates to an upper-bound on the correlation, and correspondingly a α^{-1} lower-bound on the size of the middle layer of depth 3 circuits with top AND gate and k^{-}-DNF middle gates, computing THR_r.

The corresponding hypergraph setting for the monotone question is the following:

Theorem 35. *Given a k-graph H, let f be the monotone k-DNF, with clauses formed by the edges of H,*

$$f = \bigvee_{E \in H} \bigwedge_{x \in E} x.$$

Then an assignment X is unsatisfying for f, if and only if $H \restriction_X = I_{|X|}$.

Proof of Theorem 35. Any assignment X is an unsatisfying assignment of f if and only if it false on all of the clauses of f. The clause $\bigwedge_{x \in E} x$ is true if and only if $A \subseteq X$, therefore to make X false, all the edges of H must avoid X. □

The condition that f is true on all inputs with at least $r+1$ ones, is equivalent to H being $I_{r+1}^{(k)}$-free, and the g size unsatisfying assignments are the induced $I_g^{(k)}$. The corresponding extremal problem is upper bounding $\pi\left(n, I_r^{(k)}; I_{r+1}^{(k)}\right)$.

5.3.1 Bounds for the Extremal Problems

Notice that the monotone setting is more restrictive. In particular

$$\pi\left(n, I_g^{(k)}; I_{r+1}^{(k)}\right) \leq \pi\left(n; P_{k,r}^{(g)}\right),$$

since a monotone k-DNF construction also works for the non-monotone k^{-}-DNF case.

Theorem 36. *If $k-1|r$ and $r|n(k-1)$ then* $\pi\left(n, I_r^{(k)}; I_{r+1}^{(k)}\right) \geq \frac{\binom{n(k-1)/r}{k-1}^{r/(k-1)}}{\binom{n}{r}}$.

Proof of Theorem 36. Construct a k-uniform hypergraph by splitting n into $r/(k-1)$ blocks $B_1, B_2, ..., B_{r/(k-1)}$ as equal as possible. The hypergraph attaining the bound is

$$H = \bigcup_i \binom{B_i}{k}.$$

Note any $r+1$ set must intersect one of the B_i in k or more elements by the pigeonhole principle. On the other hand, picking $k-1$ element sets from each group gives an empty I_r. This choice can be performed in $\binom{n(k-1)/r}{k-1}^{r/(k-1)}$ different ways, giving the lower-bound. □

Note that this is essentially equivalent to the construction mentioned in Section 5.2 (Theorem 15). After translating this result to circuits, with Theorem 34, the resulting DNF can be matched with the notation from Section 5.2, the DNF corresponds to the tuple

$$(k-1, k-1, k-1, ..., k-1) \in T_{\left(r/(k-1), r/(k-1), ..., r/(k-1)\right), r}.$$

Note that in the mentioned $T_{(k,k,...,k),r}$ set from Section 5.2, the DNF with the worst one-sided correlation has elements $(kr/n, kr/n, ..., kr/n)$, inverting this to get a k-DNF gives the observed difference. This is an unfortunate effect of the initial setting, but asymptotically negligible.

The hypergraph question (in a different setting) was first investigated in [HJP95]. They provide the following upper-bound.

Theorem 37. *[HJP95, Theorem 3.5]* $\pi\left(n; P_{k,r+1}^{(r)}\right) \leq \frac{k^{n-r}}{\binom{n}{r}}$

For completeness, the proof is as follows:

Proof of Theorem 37. Note that without the normalizing $\binom{n}{r}$ factor, the statement is equivalent to the upper bound on the edges, in r-graphs with property $P^{(r)}_{k,r+1}$, by k^{n-r}.

By induction on n. For $n = r$, $P^{(r)}_{k,r+1}$ is always true and the claim is trivial, both sides are 1. In general use property $P^{(r)}_{k,r+1}$ with $X = [n]$, every A in G must not fully contain $C([n])$, so it is possible to partition the edges of G into k groups by the point they miss from $C([n])$. For $x \in C([n])$ take $Y = [n] - x$ and note that the induced hypergraph $G|_Y$ also satisfies property $P^{(r)}_{k,r+1}$. So $|G_Y| \leq k^{n-r-1}$ by induction. As the k different choices for $x \in C([n])$ each provide at most that many edges, the total edge count is bounded by k^{n-r}. □

The bound is tight when $r \geq \frac{k-1}{k}n$, as it agrees with the lower-bound construction in Section 5.2. A possible way to refine this bound is the following:

Theorem 38.

$$\pi\left(n; P^{(r)}_{k,r+1}\right) \leq \frac{k^{r/(k-1)}}{\binom{rk/(k-1)}{r}}.$$

Proof of Theorem 38. Recall that $P^{(r)}_{k,r+1}$ is a hereditary hypergraph property. Each $l + r$ subset of $[n]$ satisfies $P^{(r)}_{k,r+1}$, and therefore has edge density at most $\pi\left(l + r; P^{(r)}_{k,r+1}\right) \leq \frac{k^l}{\binom{l+r}{r}}$. This gives, by averaging, the bound

$$\pi\left(n; P^{(r)}_{k,r+1}\right) \leq \min_l \frac{k^l}{\binom{l+r}{r}}.$$

Optimizing over l gives the best value is at $l = r/(k-1)$ and the bound mentioned above. □

Notice that the l appearing in the proof of Theorem 37 is optimal. The bound obtained based on [HJP95, Theorem 3.5] only considers the optimal hypergraphs up to a given size, and averages that observation, similar to the method presented in Section 2.3 without the positive semi-definite constraints. There is no insight from the global behaviour of the hypergraph. The result from Chapter 6 provides a small improvement in the monotone setting by exploiting the global behaviour of the hypergraph using the positive semi-definite relations.

This upper-bound can be translated back to circuits. Any depth 3 circuit computing r-THR with top AND gate and k^{-}-DNF middle gates, must have at least

$$\frac{\binom{rk/(k-1)}{r}}{k^{r/(k-1)}}$$

many gates in the middle layer. With k constant it is possible to use the approximation $\binom{n}{cn} = \Theta_n\left(c^{-cn}(1-c)^{-(1-c)n}(c(1-c)n)^{-1/2}\right)$ to get the bound

$$\Theta_r\left(\frac{1}{r}k^{1-\frac{r}{k(k-1)}}\left(1+\frac{1}{k-1}\right)^{rk/(k-1)}\right) \approx \exp\left(r\frac{k-\ln(k)}{(k-1)^2}\right).$$

The monotone case with $k = 2$ is the generalized Turán question, which was investigated in [Erd62, Erd84]. For larger $k > 2$, Chapter 6 gives stronger bounds, using flag algebraic techniques. In particular Theorem 19 from Chapter 6 shows that

Theorem 39. *For integers $1 < k \leq g < r$ and any $n > (r-1)\left(1+\left(\frac{(r-k)}{k-1}\right)^2\right)$,*

$$\pi\left(n, K_g^{(k)}, K_r^{(k)}\right) \leq$$
$$\leq \left(1+\frac{(r-1)(r-k)^2}{(k-1)^2 n-(r-1)\left(2k^2-2k(r+1)+r^2+1\right)}\right)\prod_{m=k}^{g}\left(1-\frac{\binom{m-1}{k-1}}{\binom{r-1}{k-1}}\right),$$

which, at the relevant parameter range and $n \to \infty$ gives the bound

$$\sqrt{2\pi r}\ \ e^{-\frac{\log(2\pi k)r}{2k}} \approx$$

$$\approx \binom{r}{k-1, k-1, \ldots, k-1} l^{-r} \leq$$

$$\leq \pi\left(K_r^{(k)}; K_{r+1}^{(k)}\right) \leq$$

$$\leq \prod_{m=k}^{r} \left(1 - \frac{\binom{m-1}{k-1}}{\binom{r}{k-1}}\right) \leq$$

$$\leq e^{1-r/k}.$$

The upper bound follows from the same compution as Section 5.2. The lower bound improves on the general non-monotone bound from Theorem 38 by a factor of $\sqrt{r/k}$, since asymptotically

$$\frac{k^{r/(k-1)}}{\binom{rk/(k-1)}{r}} \approx e^{-r/k}\sqrt{r/k}.$$

For $k \geq 3$, Theorem 38 is the best known upper bound for the extremal hypergraph problem corresponding with the non-monotone complexity of THR_r in the investigated $\binom{[n]}{r}$ slice. For a different slice of the satisfying assignments, when $k = 3$, very recently [GPP$^+$24] established a stronger bound. The non-monotone question with $k = 2$ will be completely resolved in the following theorem.

Theorem 40. *Suppose $n = ar + b$ with a, b integers and $0 \leq b < r$ (division with remainder), then*

$$\pi\left(n; P_{2,r}^{(r)}\right) = \frac{a^{r-b}(a+1)^b}{\binom{n}{r}}.$$

Note that this agrees with the construction of splitting n into r as equal groups as possible. The proof uses ideas similar to Lagrangians (see [Sau71, Mub06], where Lagrangians are used in a similar extremal hypergraph problem). In particular, a more general theorem with

weighted edge counting will be proved.

Lemma 41. *Among the r-uniform hypergraphs on n positively weighted vertices*

$$\underline{w} = (w_1, w_2, ..., w_n)$$

satisfying $P_{2,r}^{(r)}$, the maximum value of $L_{\underline{w}}(G) = \sum_{e \in G} \prod_{i \in e} w_i$ can be attained when G is r-partite.

This implies Theorem 40, as the optimal choice is to have each group as equal as possible on uniform weights.

Proof of Lemma 41. By induction on n, for $n = r$ the property $P_{2,r}^{(r)}$ is always true. Therefore, for any positive weight sequence, the single r-edge maximizes the value.

Suppose $n > r$ and G maximizes $L_{\underline{w}}(G)$. The rest will argue that an r-partite construction G' exists with the same $L_{\underline{w}}(G) = L_{\underline{w}}(G')$. Taking $X = [n]$ from the property $P_{2,r}^{(r)}$, there must be two points s, t such that no r-edge contains both s and t. Group the edges as $S = \{A - s : A \in G \wedge s \in A\}$, $T = \{A - t : A \in G \wedge t \in A\}$ and $W = \{A : A \in G \wedge \{s, t\} \cap A = \emptyset\}$. Notice that $L_{\underline{w}}(G) = L_{\underline{w}}(W) + w_s L_{\underline{w}}(S) + w_t L_{\underline{w}}(T)$. Suppose wlog that $L_{\underline{w}}(S) \geq L_{\underline{w}}(T)$, then consider $G' = W \cup \{A + s : A \in S\} \cup \{A + t : A \in S\}$. To show G' still satisfies $P_{2,r}^{(r)}$, take any $Y \subseteq [n]$ with $Y > r$. Consider the following cases based on the containment of s, t:

1. $t \notin Y$, then as G and G' agree on $[n] - t$, the property $P_{2,r}^{(r)}$ holds.
2. $t \in Y \wedge s \in Y$, then the pair $C(Y) = \{s, t\}$ works.
3. $t \in Y \wedge s \notin Y$, then as $G' \restriction_{[n]-s}$ is isomorphic to $G' \restriction_{[n]-t}$ by construction, similar to the first point, a proper $C(Y)$ set exists.

Note that G' has $L_{\underline{w}}(G') = L_{\underline{w}}(W) + (w_s + w_t) L_{\underline{w}}(S) \geq L_{\underline{w}}(G)$. This is the same as taking $L_{\underline{w}'}(W \cup \{A + s : A \in S\})$ on $[n] - t$ points but with

$$\underline{w}' = (w_1, w_2, ..., w_{s-1}, w_s + w_t, w_{s+1}, ..., w_{t-1}, w_{t+1}, ..., w_n)$$

the weight of s is replaced with $w_s + w_t$. By induction, there is an optimal construction for $[n] - t$ for any positive weights, which is r-partite. G' is constructed by duplicating a vertex in any such construction, giving that G' can be chosen as r-partite too. $\square$

It is natural to ask if there is a further relationship between $\pi\left(n; P_{k,r}^{(g)}\right)$ and $\pi\left(n, I_g^{(k)}; I_{r+1}^{(k)}\right)$ other than the relation

$$\pi\left(n, I_g^{(k)}; I_{r+1}^{(k)}\right) \leq \pi\left(n; P_{k,r}^{(g)}\right).$$

As shown by Theorem 40 and Theorem 36, when $k = 2$ the values equal

$$\pi\left(n, I_r^{(2)}; I_{r+1}^{(2)}\right) = \pi\left(n; P_{2,r}^{(r)}\right).$$

The equality also holds when $k = g$ as shown by the following theorem:

Theorem 42.

$$\pi\left(n, I_k^{(k)}; I_{r+1}^{(k)}\right) = \pi\left(n; P_{k,r}^{(k)}\right).$$

Proof. It is enough to prove that any k-graph G satisfying $P_{k,r}^{(k)}$ has complement without an induced $I_{r+1}^{(k)}$, since the $I_k^{(k)}$ counts the number of non-edges, which is the same as counting the edges in the G.

Choosing any X with size $r + 1$ in G, by property $P_{k,r}^{(k)}$, there must be some $C(X)$ with size k, which is not covered by any edge also in X. Since $C(X)$ has the same size as an edge, this means that $C(X)$ is not an edge in G, therefore X has complement not isomorphic to $I_{r+1}^{(k)}$ as required. $\square$

Contrary to the two results above, it is not true in general that equality holds in

$$\pi\left(n, I_g^{(k)}; I_{r+1}^{(k)}\right) \leq \pi\left(n; P_{k,r}^{(g)}\right).$$

[Ama23] finds small examples where this does not hold for $k \in \{3, 4, 5\}$. With a blow-up construction, the examples generate arbitrary large structures where the equality fails.

5.4 Concluding Remarks

This chapter investigated depth 3 circuits computing the threshold function. While Section 5.2 showed that under the assumption of a block structure, the construction is optimal, the question if it holds unconditionally remains open.

Conjecture 43. Π_3^k *circuits computing* THR_r *require* $|T_{(k,k,\ldots,k),r}|$ *gates in the middle layer.*

Section 5.3 formulated the question in the language of extremal hypergraph theory and attempted to give an answer with combinatorial techniques. While the $k = 2$ case is completely resolved and the bounds are tightened for larger k values, still a considerable gap remains. It was shown that $\Theta\left(\frac{r}{k}\log k\right) \geq -\log \pi\left(I_r^{(k)}; I_{r+1}^{(k)}\right) \geq -\log \pi\left(P_{k,r}^{(r)}\right) \geq \Theta\left(\frac{r}{k}\right)$, but the correct asymptotic behaviour of both functions is unknown.

Question 44. *What is the asymptotic growth of* $\pi\left(I_r^{(k)}; I_{r+1}^{(k)}\right)$ *and* $\pi\left(P_{k,r}^{(r)}\right)$*?*

Chapter 6

Generalized Turán Problem for Complete Hypergraphs

6.1 Introduction

One of the goals of this chapter is to show that the powerful plain flag algebra method can be performed by hand, resulting in a general and scaleable theorem. The main ideas and proof steps, therefore, correspond with a plain application of flag algebra and were heavily inspired by it. During the proofs, relevant parts of the flag algebra theory will be highlighted. While the asymptotic result can be fully proved with flag algebraic manipulations, the bound with finite n is only attainable with a more precise bounding of the errors.

6.1.1 Overview of the Result

In this chapter, the generalized Turán problem for complete hypergraphs will be investigated, with the following contribution:

Theorem 18. *For integers* $1 < k \leq g < r$ *and any* $n > (r-1)\left(1 + \left(\frac{(r-k)}{k-1}\right)^2\right)$,

$$\pi\left(n, K_g^{(k)}, K_r^{(k)}\right) \leq$$
$$\leq \left(1 + \frac{(r-1)(r-k)^2}{(k-1)^2 n - (r-1)\left(2k^2 - 2k(r+1) + r^2 + 1\right)}\right) \prod_{m=k}^{g} \left(1 - \frac{\binom{m-1}{k-1}}{\binom{r-1}{k-1}}\right).$$

Note this means asymptotically that

Theorem 19. *For integers* $1 < k \leq g < r$

$$\pi\left(K_g^{(k)}, K_r^{(k)}\right) \leq \prod_{m=k}^{g} \left(1 - \frac{\binom{m-1}{k-1}}{\binom{r-1}{k-1}}\right).$$

The asymptotic bound is known to be tight when $k = 2$, with the matching, balanced $(r-1)$-partite construction. Additionally, it agrees with the best-known general hypergraph Turán bound by de Caen [dC83] which is conjectured to not be tight.

[Sid95] describes various lower bound constructions for the $2 < k = g$ case. A simple construction (when $k-1$ divides $r-1$) splits the vertex set into $\frac{r-1}{k-1}$ equal groups and includes each k set that is not fully contained in a group. Using $l = \frac{r-1}{k-1}$ and an inclusion-exclusion calculation, this gives the asymptotic bound

$$\sum_{s=0}^{\lfloor g/k \rfloor} (-1)^s \binom{l}{s} \sum_{\substack{k \leq i_1, \ldots, i_s \\ i_1 + \cdots + i_s \leq g}} \binom{g}{i_1, \ldots, i_s, g - i_1 - \cdots - i_s} l^{-i_1 - \cdots - i_s} \leq \pi\left(K_g^{(k)}, K_r^{(k)}\right).$$

While it is known that this construction is not optimal when $k = g$, in the $k \ll g$ regime, where $K_g^{(k)}$ appears more if the edges are "grouped", it provides a stronger bound. Section 6.5 shows that this is asymptotically the best construction for $\pi\left(K_4^{(3)}, K_5^{(3)}\right)$. In general $g = r-1$

gives

$$\sqrt{2\pi r}\ \ e^{-\frac{\log(2\pi k)r}{2k}} \approx$$

$$\approx \binom{r-1}{k-1, k-1, \ldots, k-1} l^{-(r-1)} \leq$$

$$\leq \pi\left(K_{r-1}^{(k)}, K_r^{(k)}\right) \leq$$

$$\leq \prod_{m=k}^{r-1}\left(1 - \frac{\binom{m-1}{k-1}}{\binom{r-1}{k-1}}\right) \leq$$

$$\leq e^{(k-r)/k}.$$

Given a hypergraph G, write $d(H,G)$ for the induced density of H in G. The main tool used in the proof of Theorem 18 is:

Lemma 20. *For all $0 \leq x$ and integers $k \leq m < n$, if G is an n vertex k-graph then*

$$\begin{aligned} 0 \geq\ & \left(-\frac{1-\frac{k-1}{m}}{x}\right) && d\left(K_{m+1}^{(k)}, G\right) && + \\ & \left(2 - \frac{k-1}{mx} - \frac{1}{(n-m)x}\right) && d\left(K_m^{(k)}, G\right) && + \\ & (-x) && d\left(K_{m-1}^{(k)}, G\right). \end{aligned}$$

Note that the densities $d\left(K_m^{(k)}, G\right)$ only appear linearly in the expression. For different k, g, r parameters, a convex combination of the expressions appearing in Lemma 20, with suitable x, m values substituted, yields Theorem 18. As a comparison, [dC83] utilizes similar ideas, but with a more complicated (non-linear) expression,

$$f_{m+1} \geq \frac{m^2 f_m}{(m-k+1)(n-m)}\left(\frac{f_m(n-m+1)}{f_{m-1}m} - \frac{(k-1)(n-m)+m}{m^2}\right),$$

where $f_m = d\left(K_m^{(k)}, G\right)$ for short.

6.1.2 Outline of the Chapter

Section 6.1.3 summarizes the important notations and conventions throughout the chapter. The proof of Theorem 18 is included in Section 6.2 using two important components: Lemma 20, which is proved in Section 6.3; and a technical calculation (Lemma 46), that is included in Section 6.4. The short Section 6.5 includes a certificate for $\pi\left(K_4^{(3)}, K_5^{(3)}\right) = 3/8$. The chapter finishes with concluding remarks in Section 6.6, discussing the limitations of this approach and suggesting possible future directions.

6.1.3 Conventions

In most of the proofs, the symbol k is fixed, and the statements that appear concern k-graphs. For this reason, k superscripts from the notations are often dropped. Additionally, g, r, n symbols, with $k \leq g < r \leq n$ are reserved. They are integer parameters of the main question; bounding the value of $\pi\left(n, K_g^{(k)}, K_r^{(k)}\right)$.

The quantity

$$x_{m,r}^{(k)} = 1 - \frac{\binom{m-1}{k-1}}{\binom{r-1}{k-1}}$$

will be important, these are the terms appearing in the product. For $k = 2$, this number represent the probability that in a large $r-1$-partite graph, after having $m-1$ vertices in different parts, a randomly selected new vertex falls in another different part. Since the result is not tight for larger k, there is no probabilistic intuition for these numbers. For more details, see the discussion in Section 8.5.

6.2 Proof of Main Theorem

In this section Theorem 18 will be proved with the use of Lemma 20 and Lemma 46, a technical result included in Section 6.4. Recall the main theorem, with the introduced $x_{m,r}^{(k)}$ notation.

Theorem 18. *Given integers $1 < k \leq g < r$ and $n > (r-1)\left(1+\left(\frac{(r-k)}{k-1}\right)^2\right)$, then*

$$\pi\left(n, K_g^{(k)}, K_r^{(k)}\right)$$
$$\leq \left(1+\frac{(r-1)(r-k)^2}{(k-1)^2 n-(r-1)\left(2k^2-2k(r+1)+r^2+1\right)}\right)\prod_{m=k}^{g} x_{m,r}^{(k)}.$$

In the following text, the superscript k is dropped from the notations for easier readability. The following claim gives bounds on the $x_{m,r}$ values. It follows easily after expanding the definition of $x_{m,r}$, therefore the proof is not included.

Claim 45. *Given $k < r$ and an integer $k-1 \leq m \leq r$, then*

$$0 \leq x_{m,r}^{(k)} \leq 1,$$

with equality at $m = r$ and $m = k-1$ respectively. As a function of integer m, the smallest non-zero value the expression $x_{m,r}^{(k)}$ takes in the above range is $x_{r-1,r} = \frac{k-1}{r-1}$.

Proof of Theorem 18. Choose any $G \in \mathcal{H}_n$, and for short write $f_l = d\left(K_l, G\right)$. In the range $m \in \{k, k+1, ..., r-1\}$, $x_{m,r}$ is positive, therefore Lemma 20, with $x = x_{m,r}$ holds.

$$0 \geq -\frac{1-\frac{k-1}{m}}{x_{m,r}} f_{m+1} + \left(2-\frac{k-1}{m x_{m,r}}-\frac{1}{(n-m)x_{m,r}}\right) f_m - x_{m,r} f_{m-1}.$$

The value $(n-m)x_{m,r}$ is minimal at $m = r-1$. Use E_m for the above expression but with $\frac{1}{(n-m)x_{m,r}}$ replaced by $\frac{r-1}{(n-r+1)(k-1)}$.

$$E_m = -\frac{1-\frac{k-1}{m}}{x_{m,r}} f_{m+1} + \left(2-\frac{k-1}{m x_{m,r}}-\frac{r-1}{(n-r+1)(k-1)}\right) f_m - x_{m,r} f_{m-1}.$$

The replacement decreases the value, giving that each E_m is still non-positive. This gives that any $\underline{\delta} = (\delta_k, \delta_{k+1}, ..., \delta_{r-1})^T$ sequence with all $0 \leq \delta_m$ preserves the inequality

$$0 \geq \sum_{m=k}^{r-1} \delta_m E_m.$$

Note that each E_m (in the range $k \leq m \leq r-1$) is a linear combination of the f_l values (from the larger range $k-1 \leq l \leq r$). Therefore, the coefficient of f_l in $\sum \delta_m E_m$ depends linearly in each δ_m. Considering the f_l values as a basis, this linear dependence can be encoded in a matrix D such that $D\underline{\delta}$ is the vector of the f_l coefficients. Here D is an $(r-k+2) \times (r-k)$ matrix.

Now, assuming that the density of K_r is 0 in G gives, $f_r = d(K_r, G) = 0$. Note further that any $k-1$ vertex collection in G induces a sub-hypergraph isomorphic to K_{k-1}, since there are no edges present in both. Therefore $f_{k-1} = d(K_{k-1}, G) = 1$.

Suppose there exists $0 \leq \underline{\delta}$ such that

$$D\underline{\delta} = (-c_1, 0, 0, ..., 0, 1, 0, ..., 0, c_2)^T \tag{6.1}$$

for some c_1, c_2 values, where the 1 appears at the position corresponding with f_g. Substituting in the $f_r = 0$ and $f_{k-1} = 1$ values, it holds that

$$0 \geq \sum_{m=k}^{r-1} \delta_m E_m = -c_1 + f_g,$$

which implies the upper bound $c_1 \geq d(K_g, G)$.

Note c_1 is the coefficient of f_{k-1} which only appears in E_k. More precisely, the coefficient of f_{k-1} in E_k is $-x_{k,r}$, giving that $c_1 = x_{k,r}\delta_k$. Since the fixed coefficients in eq. (6.1) (the 0 and 1 values) do not depend on the first and last row, the relevant part of D is the middle square matrix. A description of the matrix, and the proof of the following lemma using this matrix is included in Section 6.4, with illustrative examples.

Lemma 46. *If $n > (r-1)\left(1 + \frac{(r-k)(r-1)}{(k-1)^2}\right)$, then there is a unique solution to the linear equations eq.* (6.1), *it satisfies $0 \leq \underline{\delta}$, and that*

$$\delta_k \leq \left(1 + \frac{(r-1)^2(r-k)}{(k-1)^2 n - (r-1)\left((k-1)^2 - (k-r)(r-1)\right)}\right) \prod_{m=k+1}^{g} x_{m,r}.$$

This finishes the proof of the main theorem. □

6.3 Linear Density Relations

This section covers the main combinatorial calculations involved in the proof of Lemma 20. The main purpose of Lemma 20 is to act as a building block. Not only the validity is easier to verify, but it also involves the densities $d(K_m, G)$ linearly, therefore it can be combined easily; as illustrated in Theorem 18. The small claims in this section follow closely core results from the flag algebra theory. The connection will be highlighted in Remark 50. The connection between the plain flag algebra application and this proof is discussed in Remark 51.

In the upcoming proofs, the k superscript will not be included. S is any subset of $V(G)$, while S_m is an m element subset of $V(G)$. Write $q(S)$ for the indicator function, which is 1 when S induces a complete hypergraph in G and 0 otherwise. $l(S)$ is the number of $v \in V(G) \setminus S$ where $S + v$ is complete in G. The corresponding probability is $r(S_m) = \frac{l(S_m)}{n-m}$. Similarly $rr(S)$ is the probability that two different vertex extensions are both complete.

$$rr(S_m) = \frac{\binom{l(S_m)}{2}}{\binom{n-m}{2}}.$$

When $H \in \mathcal{H}_n$, write $s(H)$ to be the number of vertices v, such that $H - v$ is isomorphic to K_{n-1}. In particular $s(K_n) = n$ and $s(K_n^-) = k$ where K_n^- represents the hypergraph on n vertices that has exactly one edge missing.

In the proof of Lemma 20, the fact that $q(S)(r(S) - x)^2$ is always non-negative, will be exploited. Understanding the terms in the square is done by the following short claims. First, $r(S)^2$ and $rr(S)$ are related.

Claim 47.

$$r(S_{m-1})^2 \leq rr(S_{m-1}) + r(S_{m-1})\frac{1}{n-m}.$$

This simply follows from

$$r(S_{m-1})^2 - rr(S_{m-1}) = r(S_{m-1})\left(\frac{l(S_{m-1})}{n-m+1} - \frac{l(S_{m-1})-1}{n-m}\right) \leq r(S_{m-1})\frac{1}{n-m}.$$

Second, linear equality between densities is shown.

Claim 48. *Suppose $m \leq l \leq n$ with $F \in \mathcal{H}_m$ and $G \in \mathcal{H}_n$ then*

$$d(F,G) = \sum_{H \in \mathcal{H}_l} d(F,H)d(H,G),$$

in particular

$$d(K_m,G) = \sum_{H \in \mathcal{H}_{m+1}} \frac{s(H)}{m+1} d(H,G).$$

Proof. Note that a uniform m sized subset of $V(G)$ can be sampled by first choosing $\mathbf{S}_l \sim \text{Unif}\binom{V(G)}{l}$ and then $\mathbf{S}_m \sim \text{Unif}\binom{\mathbf{S}_l}{m}$. The claim follows from the law of total probability. The events $\{G\restriction_{\mathbf{S}_l} \simeq H \ : \ H \in \mathcal{H}_l\}$ partition the probability space, therefore

$$\begin{aligned} d(F,G) &= \mathbb{P}\Big[G\restriction_{\mathbf{S}_m} \simeq F\Big] \\ &= \sum_{H \in \mathcal{H}_l} \mathbb{P}\Big[G\restriction_{\mathbf{S}_m} \simeq F \ \Big| \ G\restriction_{\mathbf{S}_l} \simeq H\Big]\mathbb{P}\Big[G\restriction_{\mathbf{S}_l} \simeq H\Big] \\ &= \sum_{H \in \mathcal{H}_l} d(F,H)d(H,G). \end{aligned}$$

The special case follows from $d(K_m,H) = \frac{s(H)}{m+1}$ when $H \in \mathcal{H}_{m+1}$. □

In the proof of Lemma 20, $\mathbf{S}_{m-1}$ is chosen uniformly from the possible $m-1$ sized sets. The final claim connects the expected values arising in the terms of $q(\mathbf{S}_{m-1})(p(\mathbf{S}_{m-1}) - x)^2$ with densities of various k-graphs in G.

Claim 49. *Suppose $\mathbf{S}_{m-1}$ is chosen uniformly randomly from the set $\binom{V(G)}{m-1}$ then*

1.

$$\mathbb{E}\left[q(\mathbf{S}_{m-1})r(\mathbf{S}_{m-1})\right] = d\left(K_m, G\right),$$

2.

$$\mathbb{E}\left[q(\mathbf{S}_{m-1})rr(\mathbf{S}_{m-1})\right] = \sum_{H \in \mathcal{H}_{m+1}} \frac{\binom{s(H)}{2}}{\binom{m+1}{2}} d\left(H, G\right).$$

Proof. Similar to the proof of Claim 48, first choosing $\mathbf{S}_l \sim \text{Unif}\binom{V(G)}{l}$ and then $\mathbf{S}_{m-1} \sim \text{Unif}\binom{\mathbf{S}_l}{m-1}$ results in uniformly distributed $\mathbf{S}_{m-1}$. Note that $r(\mathbf{S}_{m-1})$ and $rr(\mathbf{S}_{m-1})$ corre-

sponds to choosing 1 and 2 additional vertices accordingly, and then checking a condition on the extended set.

Let the $\mathbf{S}_m, \mathbf{S}_{m-1}$ pair be sampled as above, with $\mathbf{S}_{m-1} \subset \mathbf{S}_m$, but with uniform distribution individually. Then, conditioning on $q(\mathbf{S}_{m-1})$, the probability that $G\restriction_{\mathbf{S}_m} \simeq K_m$ is $r(\mathbf{S}_{m-1})$. Therefore, the first point follows from

$$\mathbb{E}\left[q(\mathbf{S}_{m-1})r(\mathbf{S}_{m-1})\right] = \mathbb{E}\left[G\restriction_{\mathbf{S}_m} \simeq K_m\right] = d\left(K_m, G\right).$$

For the second part, similarly sample $\mathbf{S}_{m+1}, \mathbf{S}_{m-1}$ with the properties as above. Write $RR(\mathbf{S}_{m+1}, \mathbf{S}_{m-1})$ for the event that there are two copies of K_m inside $G\restriction_{\mathbf{S}_{m+1}}$ intersecting exactly at $\mathbf{S}_{m-1}$ (note this implies $G\restriction_{\mathbf{S}_{m-1}} \simeq K_{m-1}$). Expanding using the law of total expectation gives

$$\begin{aligned}
&\mathbb{E}\left[q(\mathbf{S}_{m-1})rr(\mathbf{S}_{m-1})\right] \\
&= \sum_{H\in\mathcal{H}_{m+1}} \mathbb{P}\left[RR(\mathbf{S}_{m+1}, \mathbf{S}_{m-1}) \mid G\restriction_{\mathbf{S}_{m+1}} \simeq H\right] \mathbb{P}\left[G\restriction_{\mathbf{S}_{m+1}} \simeq H\right] \\
&= \sum_{H\in\mathcal{H}_{m+1}} \frac{\binom{s(H)}{2}}{\binom{m+1}{2}} d\left(H, G\right),
\end{aligned}$$

since in a given $G\restriction_{\mathbf{S}_{m+1}} \simeq H$, a randomly chosen $\mathbf{S}_{m-1}$ satisfies $RR(\mathbf{S}_{m+1}, \mathbf{S}_{m-1})$ with probability $\frac{\binom{s(H)}{2}}{\binom{m+1}{2}}$. □

Remark 50. *The above claims all correspond to parts of the general flag algebra theory [Raz07].*

1. *Claim 48 corresponds to the chain rule (Lemma 2.2 in [Raz07]). In the language of flags it gives*
$$K_m = \sum_{H\in\mathcal{H}_{m+1}} \frac{s(H)}{m+1} H.$$
2. *Claim 47 corresponds to products (Lemma 2.3 in [Raz07]). It is more or less equivalent with*
$$p\left(K_m^{T_{m-1}}; G^{T_{m-1}}\right)^2 - p\left(K_m^{T_{m-1}}, K_m^{T_{m-1}}; G^{T_{m-1}}\right) = O(|G|^{-1}).$$

3. Claim 49 corresponds to averaging (Theorem 2.5 in [Raz07]). It is a restatement of

$$\left[\!\left[K_m^{T_{m-1}}\right]\!\right]_{T_{m-1}} = K_m$$

and

$$\left[\!\left[\left(K_m^{T_{m-1}}\right)^2\right]\!\right]_{T_{m-1}} = \sum_{H \in \mathcal{H}_{m+1}} \frac{\binom{s(H)}{2}}{\binom{m+1}{2}} H.$$

With these claims, the main lemma follows easily.

Lemma 20. *For all $x > 0$ and integers $k \leq m < n$, if $G \in \mathcal{H}_n$ then the following holds*

$$\begin{aligned} 0 \geq \; & \left(-\frac{1-\frac{k-1}{m}}{x}\right) & & d\left(K_{m+1}, G\right) & + \\ & \left(2 - \frac{k-1}{mx} - \frac{1}{(n-m)x}\right) & & d\left(K_m, G\right) & + \\ & (-x) & & d\left(K_{m-1}, G\right). \end{aligned}$$

Proof. When $x \in \mathbb{R}$ the quantity $q(S)(r(S) - x)^2$ is always non-negative. Therefore choosing $\mathbf{S}_{m-1} \sim \text{Unif} \binom{V(G)}{m-1}$ the following is true

$$0 \leq \mathbb{E}\left[q(\mathbf{S}_{m-1})(r(\mathbf{S}_{m-1}) - x)^2\right].$$

Expanding the terms and applying Claim 47 gives

$$0 \leq \mathbb{E}\left[q(\mathbf{S}_{m-1}) r r(\mathbf{S}_{m-1}) + \left(\frac{1}{n-m} - 2x\right) q(\mathbf{S}_{m-1}) r(\mathbf{S}_{m-1}) + x^2 q(\mathbf{S}_{m-1})\right].$$

The substitution from Claim 49 yields the following expression, without expected values:

$$0 \leq \sum_{H \in \mathcal{H}_{m+1}} \frac{\binom{s(H)}{2}}{\binom{m+1}{2}} d\left(H, G\right) + \left(\frac{1}{n-m} - 2x\right) d\left(K_m, G\right) + x^2 d\left(K_m, G\right).$$

Notice that $s(H)$ is maximal on K_{m+1}, otherwise it is at most k. This observation gives

$$\begin{aligned}
0 \leq \frac{k-1}{m} & \sum_{\substack{H \in \mathcal{H}_{m+1} \\ H \neq K_{m+1}}} \frac{s(H)}{(m+1)} d\left(H, G\right) \\
& + d\left(K_{m+1}, G\right) \\
& + \left(\frac{1}{n-m} - 2x\right) d\left(K_m, G\right) \\
& + x^2 d\left(K_{m-1}, G\right).
\end{aligned}$$

Finally expanding $\frac{k-1}{m} d(K_m, G)$ using Claim 48 results in

$$\begin{array}{llll}
0 \leq & \left(1 - \frac{k-1}{m}\right) & d\left(K_{m+1}, G\right) & + \\
& \left(\frac{k-1}{m} + \frac{1}{(n-m)} - 2x\right) & d\left(K_m, G\right) & + \\
& x^2 & d\left(K_{m-1}, G\right). &
\end{array}$$

Since $x \geq 0$, note that Lemma 20 is a $-1/x$ multiple of the above, and the proof is complete. □

Remark 51. *Lemma 20 can be easily stated as*

$$0 \leq \left(1 - \frac{k-1}{m}\right) K_{m+1} + \left(\frac{k-1}{m} - 2x\right) K_m + x^2 K_{m-1}$$

in the language of flags. The proof uses the expansion of the simple square

$$0 \leq \left[\!\left[\left(K_m^{T_{m-1}} - x T_{m-1}\right)^2 \right]\!\right]_{T_{m-1}}.$$

In a plain application of flag algebra, the computer finds a conic combination of squares, similar to the above expression. Lemma 20 provides squares in a form that is easy to handle later (they only involve a small number of $d(K_m, G)$ values). The following section shows that the target expression

$$K_g \leq \prod_{m=k}^{g} x_{m,r}^{(k)} + c K_r,$$

for some c constant, lies in the conic combination of the squares.

6.4 The Associated Tridiagonal Matrix

Recall the matrix equation in eq. (6.1),

$$D\underline{\delta} = (-c_1, 0, 0, ..., 0, 1, 0, ..., 0, c_2)^T.$$

For the rest of this section, all g values are simultaneously considered, they only change the location of the 1 in $(-c_1, 0, 0, ..., 0, 1, 0, ..., 0, c_2)^T$. As outlined in the proof of theorem 18, the first and last row of the matrix D is not used, when solving for the 0 and 1 values. For given k, r values, the resulting $(r-k) \times (r-k)$ square matrix will be called the associated tridiagonal matrix.

Definition 52. *Given $k < r$ integers, the associated tridiagonal matrix is $D_r^{(k)}$. Its entries are indexed by the range $k \leq l, m < r$, and equal to*

$$d_{l,m} = \begin{cases} -x_{m,r}^{(k)} & \text{if } l = m-1 \\ 2 - \frac{k-1}{m x_{m,r}^{(k)}} & \text{if } l = m \\ -\frac{1-\frac{k-1}{m}}{x_{m,r}^{(k)}} & \text{if } l = m+1 \\ 0 & \text{otherwise.} \end{cases}$$

Recall the expression for E_m,

$$E_m = -\frac{1-\frac{k-1}{m}}{x_{m,r}} f_{m+1} + \left(2 - \frac{k-1}{m x_{m,r}} - \frac{r-1}{(n-r+1)(k-1)}\right) f_m - x_{m,r} f_{m-1}.$$

The row and column indexing in the definition of $D_r^{(k)}$ is shifted to match with the indexing

from E_m. In particular, the (l, m) element in the associated tridiagonal matrix agrees with the coefficient of f_l in E_m, after taking $n \to \infty$. This makes the indexing start from $(l, m) = (k, k)$ and end at $(r-1, r-1)$. Below is an example of the matrix $D_6^{(2)}$.

$$D_6^{(2)} = \begin{pmatrix} d_{2,2} & d_{2,3} & d_{2,4} & d_{2,5} \\ d_{2,3} & d_{3,3} & d_{3,4} & d_{3,5} \\ d_{2,3} & d_{4,3} & d_{4,4} & d_{4,5} \\ d_{2,3} & d_{5,3} & d_{5,4} & d_{5,5} \end{pmatrix} = \begin{pmatrix} 11/8 & -3/5 & 0 & 0 \\ -5/8 & 13/9 & -2/5 & 0 \\ 0 & -10/9 & 11/8 & -1/5 \\ 0 & 0 & -15/8 & 1 \end{pmatrix}.$$

Since the target vectors in $D_r^{(k)}\underline{\delta} = (0, 0, ..., 0, 1, 0, ..., 0)^T$ are the unit vectors, the solution for the gth value is the same as the gth column of the inverse of $D_r^{(k)}$. It will be shown that the matrix $D_r^{(k)}$ has full rank and therefore has an inverse. Suppose $\Delta_r^{(k)}$ is the inverse of $D_r^{(k)}$, with entries $\delta_{m,g}$ indexed using the same range. Continuing the same $(k, r) = (2, 6)$ example yields

$$\Delta_6^{(2)} = \begin{pmatrix} \delta_{2,2} & \delta_{2,3} & \delta_{2,4} & \delta_{2,5} \\ \delta_{3,2} & \delta_{3,3} & \delta_{3,4} & \delta_{3,5} \\ \delta_{4,2} & \delta_{4,3} & \delta_{4,4} & \delta_{4,5} \\ \delta_{5,2} & \delta_{5,3} & \delta_{5,4} & \delta_{5,5} \end{pmatrix} = \begin{pmatrix} 1 & 3/5 & 6/25 & 6/125 \\ 5/8 & 11/8 & 11/20 & 11/100 \\ 25/36 & 55/36 & 29/18 & 29/90 \\ 125/96 & 275/96 & 145/48 & 77/48 \end{pmatrix}.$$

Lemma 46 has two claims about this inverse. Firstly, to preserve the direction of the inequalities from lemma 20, all $\delta_{m,g}$ entries must be positive. Secondly, it claims an upper bound on the $\delta_{k,g}$ values. In particular, the proof of theorem 18 shows at the limit that $\pi(K_g, K_r) \leq x_{k,r}\delta_{k,g}$, and correspondingly an upper bound on $\delta_{k,g}$ translates to an upper bound on $\pi(K_g, K_r)$.

To illustrate this on the $(k, r) = (2, 6)$ example, by inspection, the entries of $\Delta_6^{(2)}$ are positive, therefore the $x_{2,6}^{(2)} = \frac{4}{5}$ multiple of the first row translates to the bounds

$$\pi(K_2, K_6) \leq \frac{4}{5},\ \pi(K_3, K_6) \leq \frac{12}{25},\ \pi(K_4, K_6) \leq \frac{24}{125},\ \pi(K_5, K_6) \leq \frac{24}{625}.$$

Write $\epsilon = \frac{r-1}{(n-r+1)(k-1)}$ and notice that $D_r^{(k)} - \epsilon I$ has column values equal to the coefficients in E_m from theorem 18 (without taking the limits). Using this new notation, recall lemma 46.

Lemma 46. *If $0 \leq \epsilon < \frac{k-1}{(r-1)(r-k)}$, then the matrix $\left(D_r^{(k)} - \epsilon I\right)$ is invertible. Call $\Delta_r^{(k)}(\epsilon) = \left(D_r^{(k)} - \epsilon I\right)^{-1}$ the inverse of this matrix, with entries $\delta_{m,g}(\epsilon)$. The values $\delta_{m,g}(\epsilon)$ are all positive, and the first row is bounded by*

$$\delta_{k,g}(\epsilon) \leq \frac{1}{1 - \epsilon\frac{(r-1)(r-k)}{k-1}} \prod_{m=k}^{g} x_{m+1,r}.$$

The case $\epsilon = 0$ corresponds to the inverse of $D_r^{(k)}$, the asymptotic question when $n \to \infty$. This section is devoted to the proof of lemma 46, but it is illuminating and helpful for the proof to first prove the corresponding statement about $D_r^{(k)}$.

6.4.1 The inverse of $D_r^{(k)}$

Lemma 53. *The associated tridiagonal matrix is invertible. Let $\Delta_r^{(k)} = \left(D_r^{(k)}\right)^{-1}$ be the inverse, with entries $\delta_{m,g}$. Then $\delta_{m,g}$ are all positive and*

$$\delta_{k,g} = \prod_{m=k+1}^{g} x_{m,r}^{(k)}.$$

In the following proofs, the k superscripts are omitted to increase readability. The complete inverse can be calculated following the method described in [Usm94]. The value θ_m represents the determinant of the rows and columns indexed by the set $\{k, k+1, ..., m\}$, while ϕ_m is the determinant for the rows and columns indexed by $\{m, m+1, ..., r-1\}$. The next claim is a relabeling of [Usm94]. It follows from the cofactor expansion of the determinant, the proof is not included here.

Claim 54.

1. *For all $k \leq m < r$ the induction*

$$\theta_m = d_{m,m}\theta_{m-1} - d_{m-1,m}d_{m,m-1}\theta_{m-2}$$

holds with initial values $\theta_{k-1} = 1$ *and* $\theta_{k-2} = 0$.

2. *For all* $k \leq m < r$ *the reverse induction*

$$\phi_m = d_{m,m}\phi_{m+1} - d_{m+1,m}d_{m,m+1}\phi_{m+2}$$

holds with initial values $\phi_r = 1$ *and* $\phi_{r+1} = 0$.

3. *For all* $k \leq m < r$ *the determinant can be calculated*

$$\mathrm{Det}\,(D_r) = \theta_{r-1} = \phi_k = \theta_m\phi_{m+1} - d_{m,m+1}d_{m+1,m}\theta_{m-1}\phi_{m+2}.$$

4. *The entries of the inverse matrix are*

$$\delta_{m,g} = \frac{(-1)^{m+g+r-k}}{\mathrm{Det}(D_r)} \begin{cases} \theta_{m-1}\phi_{g+1}\prod_{i=m}^{g-1} d_{i,i+1} & \text{if } m \leq g \\ \\ \theta_{g-1}\phi_{m+1}\prod_{i=g}^{m-1} d_{i+1,i} & \text{otherwise,} \end{cases}$$

in the corresponding $k \leq m, g < r$ *range.*

First, the ϕ_m values will be calculated using the recursive expression above.

Claim 55. $\phi_m = 1$ *in the range* $k \leq m \leq r$.

Proof. By reverse induction. The claim holds for $m = r$ and notice

$$\phi_{r-1} = d_{r-1,r-1} = 2 - \frac{\frac{k-1}{r-1}}{1 - \frac{\binom{r-2}{k-1}}{\binom{r-1}{k-1}}} = 1.$$

Using point 2 from claim 54

$$\begin{aligned}
\phi_m &= d_{m,m} - d_{m+1,m} d_{m,m+1} \\
&= 2 - \frac{k-1}{m x_{m,r}} - \left(1 - \frac{k-1}{m}\right) \frac{x_{m+1,r}}{x_{m,r}} \\
&= 2 - \frac{k-1}{m(1-u)} - \left(1 - \frac{k-1}{m}\right) \frac{1 - \frac{mu}{m-k+1}}{1-u} \\
&= 1,
\end{aligned}$$

where $u = \frac{\binom{m-1}{k-1}}{\binom{r-1}{k-1}} = 1 - x_{m,r}$ and $u\frac{m}{m-k+1} = \frac{\binom{m}{k-1}}{\binom{r-1}{k-1}} = 1 - x_{m+1,r}$ substitution was used to simplify the calculation. □

This gives that $\text{Det}\,(D_r) = \theta_{r-1} = \phi_k = 1$, and demonstrates that the associated tridiagonal matrix is invertible.

Claim 56. *In the* $k-1 \leq m < r$ *range,* $\text{Sign}(\theta_m) = 1$.

Proof. By induction, note that the claim holds for $\theta_{k-1} = 1$. Then using point 3 from claim 54 and the value for the determinant,

$$\theta_m = 1 + d_{m,m+1} d_{m+1,m} \theta_{m-1}.$$

Using 45, note that the values $d_{m,m+1} = -x_{m+1,r}$ and $d_{m+1,m} = -\frac{1-\frac{k-1}{m}}{x_{m,r}}$ are both negative. Therefore their product; and by induction, θ_{m-1}, are positive. □

Proof of lemma 53. It claims two things,

1. The entries $\delta_{m,g}$ are all positive: claim 54 point 4 gives that

$$\text{Sign}(\delta_{m,g}) = (-1)^{m+g+r-k} \begin{cases} \prod_{i=m}^{g-1} \text{Sign}(d_{i,i+1}) & \text{if } m \leq g \\ \prod_{i=g}^{m-1} \text{Sign}(d_{i+1,i}) & \text{otherwise,} \end{cases}$$

   using $\text{Sign}(\theta_m) = 1$ from claim 56 and that $\phi_m = 1$ from claim 55. Since all $d_{i,i+1}, d_{i+1,i}$ are negative, the inverse of D_r has only positive entries.

2. $\delta_{k,g} = \prod_{m=k}^{g} x_{m+1,r}$: Again substituting the values $d_{i,i+1} = -x_{i+1,r}$ and $\text{Det}(D_r) = \phi_{g+1} = \theta_{k-1} = 1$ into claim 54 point 4 gives the stated value for $\delta_{k,g}$.

□

Interestingly, the values $\delta_{k,g}$ are easy enough to calculate exactly. In contrast, for $m > k$, the entries $\delta_{m,g}$ require the value of θ_m which is difficult to find in general with the recursive expression. Surprisingly, the sign of θ_m is easy to find, exactly what is needed for the proof.

6.4.2 The inverse of $D_r^{(k)} - \epsilon I$

Consider the same calculation but with $D_r - \epsilon I$. The value $\phi_m(\epsilon)$ is the determinant for the rows and columns indexed by $\{m, m+1, ..., r-1\}$ of $D_r - \epsilon I$. The next claim shows that $\phi_m(\epsilon)$ is increasing in m when $\epsilon > 0$. A notation for the increments will be useful, write $\zeta_m(\epsilon) = \phi_{m+1}(\epsilon) - \phi_m(\epsilon)$.

Claim 57. *When $k \leq m < r$ and $0 \leq \epsilon \leq \frac{k-1}{(r-1)(r-m)}$,*

$$0 \leq \zeta_m(\epsilon) \leq \epsilon \frac{r-1}{k-1} \left(1 - \left(1 - \frac{k-1}{r-1} \right)^{r-m} \right)$$

and correspondingly

$$1 \geq \phi_m(\epsilon) \geq 1 - \epsilon \frac{(r-1)(r-m)}{k-1}.$$

Proof. Use point 2 from claim 54. The initial value is $\zeta_r(\epsilon) = 0$ and by reverse induction take

$$\begin{aligned}
\phi_m(\epsilon) &= \left(d_{m,m} - \epsilon\right) \phi_{m+1}(\epsilon) - d_{m+1,m} d_{m+1,m} \phi_{m+2}(\epsilon) \\
&= \left(2 - \frac{k-1}{m x_{m,r}} - \epsilon \right) \phi_{m+1}(\epsilon) - \left(1 - \frac{k-1}{m x_{m,r}} \right) \left(\phi_{m+1}(\epsilon) + \zeta_{m+1}(\epsilon) \right) \\
&= \phi_{m+1}(\epsilon) - \zeta_{m+1}(\epsilon) \left(1 - \frac{k-1}{m x_{m,r}} \right) - y \phi_{m+1}(\epsilon).
\end{aligned}$$

Therefore

$$\zeta_m(\epsilon) = \zeta_{m+1}(\epsilon) \left(1 - \frac{k-1}{m x_{m,r}} \right) + \epsilon \phi_{m+1}(\epsilon). \tag{6.2}$$

Note that

$$0 \le d_{m,m+1} d_{m+1,m} = \left(1 - \frac{k-1}{m x_{m,r}}\right) \le \left(1 - \frac{k-1}{r-1}\right),$$

in the $k \le m < r$ range. As $\epsilon \le \frac{k-1}{(r-1)(r-m)} < \frac{k-1}{(r-1)(r-m-1)}$ by reverse induction it holds that $0 \le \phi_{m+1}(\epsilon)$ and $0 \le \zeta_{m+1}(\epsilon)$ giving the required lower bound $0 \le \zeta_m(\epsilon)$. This implies the upper bound $\phi_m(\epsilon) \le 1$.

For the $\zeta_m(\epsilon)$ upper bound, in eq. (6.2) bound each term: $m x_{m,r} \le (r-1)$ and $\phi_{m+1}(\epsilon) \le 1$. This gives the intermediate result

$$\zeta_m(\epsilon) \le \zeta_{m+1}(\epsilon) \left(1 - \frac{k-1}{r-1}\right) + \epsilon.$$

which, by iterated application and $\zeta_r(\epsilon) = 0$ initial value, implies

$$\zeta_m(\epsilon) \le \epsilon \frac{r-1}{k-1} \left(1 - \left(1 - \frac{k-1}{r-1}\right)^{r-m}\right).$$

A summation formula for the upper and lower $\zeta_m(\epsilon)$ bounds combined with the initial $\phi_r(\epsilon) = 1$ value gives

$$1 \ge \phi_m(\epsilon) \ge 1 - \epsilon \frac{(r-1)^2}{(k-1)^2} \left(\frac{(k-1)(r-m)}{r-1} + \left(1 - \frac{k-1}{r-1}\right)^{r-m} - 1\right).$$

This provides a tighter bound, but for simplicity use

$$1 - \epsilon \frac{(r-1)^2}{(k-1)^2} \left(\frac{(k-1)(r-m)}{r-1} + \left(1 - \frac{k-1}{r-1}\right)^{r-m} - 1\right) \ge 1 - \epsilon \frac{(r-1)(r-m)}{k-1}.$$

□

From the above, a simple linear bound for the determinant can be deduced. When $0 < \epsilon < \frac{k-1}{(r-1)(r-k)}$,

$$1 - \epsilon \frac{(r-1)(r-k)}{k-1} \le \phi_k(\epsilon) = \mathrm{Det}(D_r - \epsilon I) \le 1.$$

If $0 \le \epsilon < \frac{k-1}{(r-1)(r-k)}$, then the determinant is strictly positive, making $\mathrm{Det}(D_r - \epsilon I)$ invertible,

while also bounding the smallest eigenvalue of D_r away from 0.

Proof of lemma 46. Two main properties are required from the inverse.

1. The entries $\delta_{m,g}(\epsilon)$ are all positive: By assumption, ϵ is smaller than the smallest eigenvalue of D_r. Therefore the expansion

$$(D_r - \epsilon I)^{-1} = D_r^{-1} + \epsilon D_r^{-2} + \epsilon^2 D_r^{-3}...$$

holds. Lemma 53 shows that the entries in D_r^{-1} (and in D_r^{-i} for $1 > i$ correspondingly) are all positive, giving the required positivity of $\delta_{m,g}(\epsilon)$.

2.
$$\delta_{k,g}(\epsilon) \leq \frac{1}{1 - \epsilon\frac{(r-1)(r-k)}{k-1}} \prod_{m=k}^{g} x_{m+1,r} :$$

This follows from substituting the bounds $\phi_m(\epsilon) \leq 1$ and $1 - \epsilon\frac{(r-1)(r-k)}{k-1} \leq \text{Det}(D_r - \epsilon I)$ into 54 point 4.

□

6.5 $\pi\left(K_4^{(3)}, K_5^{(3)}\right) = 3/8$

The combination of more sophisticated, but still simple squares can provide tight bounds for $\pi\left(K_4^{(3)}, K_5^{(3)}\right) = 3/8$. For an easier description of the flags, consider the complement question. If E_n is the 3-graph with n vertices and no edges then

$$\pi\left(K_4^{(3)}, K_5^{(3)}\right) = \lim_{n\to\infty} \max\left\{d(E_4, G) \ : \ G \in \mathcal{H}_n^{(3)}, \ \ d(E_5) = 0\right\}.$$

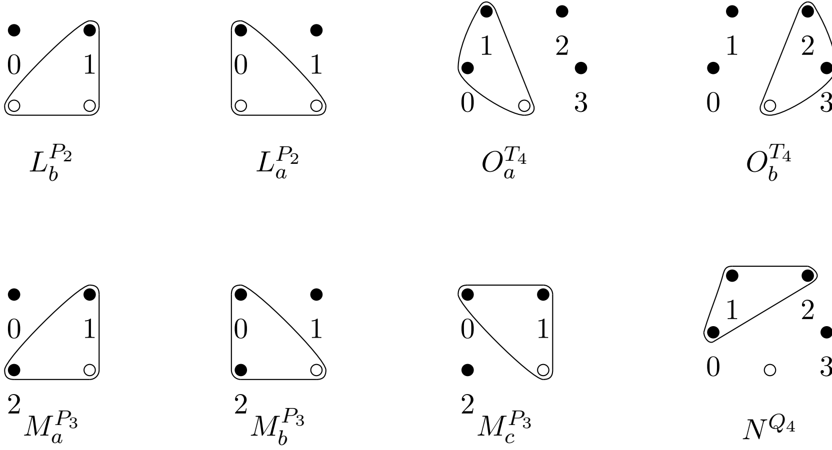
Use P_n for the corresponding type with n vertices and no edges. Note that $E_n^{P_m}$ is unique for any $m < n$ pair. Define the further flags:

1. $L_a^{P_2}$ is the flag with vertex set $\{0, 1, 2, 3\}$, edge set $\{(0, 2, 3)\}$ and type formed from the vertices $0, 1$. Similarly, write $L_b^{P_2}$ for the flag with the same vertex set and type but $\{(1, 2, 3)\}$ edge set.

2. Use $M_a^{P_3}$ for the flag with vertices $\{0,1,2,3\}$, edges $\{(1,2,3)\}$ and type from $0,1,2$. Symmetrically, with the same vertex and type set use $M_b^{P_3}$ for the edge set $\{(0,2,3)\}$. And $M_c^{P_3}$ for the edge set $\{(0,1,3)\}$.

3. N^{Q_4} is the flag with vertices $\{0,1,2,3,4\}$, edges $\{(0,1,2)\}$ and type formed by $0,1,2,3$. Note that Q_4 does not agree with any of the T_n or P_n types.

4. $O_a^{T_4}$ has vertex set $\{0,1,2,3,4\}$, edge set $\{(0,1,4)\}$ and type formed from $0,1,2,3$. Additionally write $O_b^{T_4}$ for the flag with the same vertex and type set but $\{(2,3,4)\}$ edges.



Then the following inequality holds on hypergraphs without induced copies of E_5:

$$\begin{aligned} 0 \leq\ & \frac{2}{3}\left[\!\!\left[\left(E_3^{P_1} - \frac{3}{4}P_1\right)^2\right]\!\!\right]_{P_1} + \frac{1}{6}\left[\!\!\left[\left(L_a^{P_2} - L_b^{P_2}\right)^2\right]\!\!\right]_{P_2} + \\ & \frac{13}{12}\left[\!\!\left[\left(M_a^{P_3} + M_b^{P_3} + M_c^{P_3} - \frac{1}{2}P_3\right)^2\right]\!\!\right]_{P_3} + \frac{11}{12}\left[\!\!\left[\left(E_4^{P_3} - \frac{1}{2}P_3\right)^2\right]\!\!\right]_{P_3} + \\ & 2\left[\!\!\left[\left(N^{Q_4} - \frac{1}{2}Q_4\right)^2\right]\!\!\right]_{Q_4} + \frac{1}{2}\left[\!\!\left[\left(O_a^{T_4} - O_b^{T_4}\right)^2\right]\!\!\right]_{P_4} \leq \frac{3}{8} - E_4. \end{aligned} \tag{6.3}$$

So far, the only verification of Equation (6.3) requires a tedious (computer-assisted) checking of all the 2102 hypergraphs in $\mathcal{H}_6^{(3)}$ without E_5. The implementation of flag algebras in the sage mathematical software can verify this claim. The corresponding lower bound is attained at $G_n = K^{(3)}_{\lfloor n/2 \rfloor} \bigsqcup K^{(3)}_{\lceil n/2 \rceil}$. Note that $d(E_5, G_n) = 0$ while $\lim_{n\to\infty} d(E_4, G_n) = \frac{3}{8}$.

6.6 Concluding Remarks

This chapter investigated a natural extension of the generalized Turán problem to hypergraphs. The result matches the best-known general bounds for k-graphs but fails to provide tight bounds when $k > 3$.

The main combinatorial insight comes from the simple inequality

$$0 \leq \left[\!\left[\left(K_m^{T_{m-1}} - xT_{m-1} \right)^2 \right]\!\right]_{T_{m-1}}, \tag{6.4}$$

combined with a close approximation of $\left[\!\left[\left(K_m^{T_{m-1}} \right)^2 \right]\!\right]_{T_{m-1}}$. As shown in the chapter, the convex combination of these squares includes difficult results for the generalized hypergraph Turán problem.

The long list of questions improved by the plain flag algebraic method indicates that finding more sophisticated squares can greatly improve the available density bounds. It would be interesting to identify other families of simple linear density relations (like the one described in Lemma 20) whose conic combination includes new bounds for extremal hypergraph problems, even better if the bounds are tight. The provided certificate for $\pi\left(K_4^{(3)}, K_5^{(3)}\right) = 3/8$ can perhaps be generalized to larger cases. It is interesting that for the smallest k, g, r tuple, which is not already known ($k = 2$) and is not a classical hypergraph Turán problem ($k = g$), the exact solution follows from flag algebraic calculations. It also highlights the limitations of computer assisted searches: calculations for problems with higher parameters are infeasible.

6.6.1 Finding Squares

There is an easy to describe reason why Equation (6.4) fails to provide tight bounds for k-graphs where $k > 2$ but is asymptotically exact when $k = 2$. The extremal configuration for $\pi\left(n, K_g^{(2)}, K_r^{(2)}\right)$ is a unique balanced $(r-1)$-partite graph, call it $G_r(n)$, and say $G_r^{T_{m-1}}(n)$ is the same structure with a complete $(m-1)$-tuple marked as a type. Any choice of T_{m-1} results in the same $\lim_{n\to\infty} d\left(K_m^{T_{m-1}}, G_r^{T_{m-1}}(n)\right)$ value, which is $x_{m,r}^{(2)} = 1 - \frac{m-1}{r-1}$. In contrast, the conjectured optimal constructions when $k > 2$ give different values for different T_{m-1} choices;

therefore, no $x \in \mathbb{R}$ exists with

$$\left[\!\left[\left(K_m^{T_{m-1}} - xT_{m-1} \right)^2 \right]\!\right]_{T_{m-1}} = 0$$

on the conjectured optimal constructions. This slackness gives the difference between the conjectured optimal constructions and the proved bounds here. The values $x_{m,r}^{(k)}$ are chosen optimally, any asymptotically significant improvement must utilize a different combinatorial insight.

Chapter 7

Flag Algebra Calculus in SAGE

Appendix A contains the code written to perform flag algebraic calculations accompanying the results of this thesis. This chapter contains details about the implementation, example usage of the code and interesting results it can prove. The code extends the SAGE mathematical software [The24], implementing flag algebraic objects and standard calculations on them.

7.1 SDP and Flag Algebras

In a fixed combinatorial theory T, write the flags of size n and type τ as $\mathcal{H}_n^\tau$. The multiplication table is $\mathcal{H}_{n'}^\tau \otimes \mathcal{H}_{n'}^\tau$, which results in $2n' - |\tau|$ sized flags. After averaging it becomes

$$\llbracket \mathcal{H}_{n'}^\tau \otimes \mathcal{H}_{n'}^\tau \rrbracket_\tau$$

which is a matrix formed from linear combinations of $\mathcal{H}_{2n'-|\tau|}$ elements. The matrix formed from the coefficients of flag $F \in \mathcal{H}_n$ is

$$M_F^{\tau,n} := d\left(\llbracket \mathcal{H}_{n'}^\tau \otimes \mathcal{H}_{n'}^\tau \rrbracket_\tau, F \right).$$

Here $2n' - |\tau| = n$, so the result is a linear combination of $\mathcal{H}_n$ flags. In particular, the multiplication table $M^{\tau,n}$ has dimensions

$$M^{\tau,n} \in \mathbb{R}^{|\mathcal{H}_n| \times |\mathcal{H}^{\tau}_{n'}| \times |\mathcal{H}^{\tau}_{n'}|}.$$

The algebra homomorphisms $\Phi = \lim_{n\to\infty} d(-, G_n) : \mathcal{H}_n \longrightarrow \mathbb{R}$ must satisfy the following (on the small slice of $\mathcal{H}_n$):

$$\begin{aligned} \forall \tau : \sum_{F \in \mathcal{H}_n} M^{\tau,n}_F \Phi(F) \succeq 0 \\ \forall F \in \mathcal{H}_n : \Phi(F) \geq 0 \\ \sum_{F \in \mathcal{H}_n} \Phi(F) = 1. \end{aligned}$$

In the standard setting, the goal is to maximize a given target flag's (say T) density. In that case the optimization problem can be easily expressed as

$$\begin{aligned} \max_{\Phi \in \mathbb{R}^{\mathcal{H}_n}} \quad & \sum_F d(T, F)\Phi(F) \\ \text{s.t.} \quad & \forall \tau : \sum_{F \in \mathcal{H}_n} M^{\tau,n}_F \Phi(F) \succeq 0 \\ & \forall F \in \mathcal{H}_n : \Phi(F) \geq 0 \\ & \sum_{F \in \mathcal{H}_n} \Phi(F) = 1. \end{aligned}$$

This is exactly an SDP problem, and can be approximated with out-of-the-box SDP solvers. Constraints of the form $d(G, -) \geq d$ can be simply added to the optimization problem as $\sum_F d(G, -)\Phi(F) \geq d$, resulting only in an additional linear constraint.

7.2 Implementation Details

For a fixed theory T, provided that the models form a hereditary property, as outlined in Section 2.3, the $\{\lim_{n\to\infty} d(H, -)\}_{H \in \mathcal{H}}$ objects and their linear combinations form an algebra over any ring that contains the rationals.

The implementation contains four major classes:

1. `CombinatorialTheory`: to capture theories T and their models $\mathcal{H}$,
2. `Flag`: to represent elements, flags and types from a given `CombinatorialTheory`,
3. `FlagAlgebra`: to represent the entire algebra $\mathcal{A}^T$ formed from the linear combination of `Flag`, with a given type T in a `CombinatorialTheory`,
4. `FlagAlgebraElement`: to represent elements of a given `FlagAlgebra`.

7.2.1 `CombinatorialTheory`

A combinatorial theory is any theory whose models satisfy a hereditary property (recall Section 2.3), which is a sufficient condition to define a flag algebra over the elements of the theory. This means that functional symbols are not allowed in the theory, only relational symbols. To construct a `CombinatorialTheory`, the following is required:

- `name`: the name of the theory (for example `GraphTheory`),
- `signature`: the signature of the relational symbols in the theory, with their names (for example `edges=2`),
- `generator`: a function that generates non-equal elements of the theory with a given size,
- `identifier`: a function that creates a unique identifier for any model in the theory, such that equal (isomorphic) models get equal identifiers.

The code contains pre-implemented `CombinatorialTheory` objects. For regular graphs, 3-graphs, directed graphs, tournaments, permutations, graphs with ordered edges, graphs with ordered vertices and graphs with two colored edges (for Ramsey style calculations) all have a corresponding `CombinatorialTheory` object (`GraphTheory`, `ThreeGraphTheory`, `DiGraphTheory`, `TournamentTheory`, `PermutationTheory`, `OEGraphTheory`, `OVGraphTheory`, `RamseyGraphTheory` respectively).

In addition, the `CombinatorialTheory` objects can be refined, by specifying a collection of models to exclude. Given a theory `T`, and a list of models `model_list = [H_0, H_1, ...]`, the theory `T.exclude(model_list)` contains exactly the models of `T` that satisfy the hereditary property $P_{\text{model_list}}$ (so models without induced substructure from `model_list`).

Other than the generation and identification of the models in the theory, the class can also construct the multiplication table, as defined in Section 7.1. A simple memorization is implemented in the `CombinatorialTheory` class, as the generated data (the list of models and the multiplication table) are usually hard to compute, but are frequently needed in later calculations.

7.2.2 `Flag`

In the implementation, `Flag` acts as the models of a given theory, the types and the flags that one can construct from the models. In particular, every `Flag` is an element of a `CombinatorialTheory` object. Furthermore, a model is identified with a flag with empty type. Since the word `type` is used in Python for checking the class of an object, the type in flag algebra is represented by `ftype` in the code.

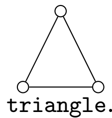
To construct a `Flag` the following is required:

- `theory`: the theory the flag is model of,
- `n`: the size of the vertex set, with $[n]$ as the list of vertices,
- the sets for each relation symbol in the signature (if not provided then the signature is assumed to be empty),
- `ftype`: the list of vertices forming the type (if not provided then the type is assumed to be empty).

As a shortcut, calling a theory constructs an element `Flag` from the theory with the same parameters. Creating a triangle graph therefore could be performed with both `triangle = Flag(GraphTheory, 3, edges=[[0, 1], [0, 2], [1, 2]])`,

`triangle = GraphTheory(3, edges=[[0, 1], [0, 2], [1, 2]])`.

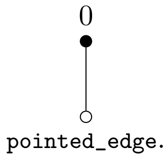
They both produce K_3



It is possible to provide a type to the flags, by assigning a list of vertices to `ftype` as follows:

```
pointed_edge = Flag(GraphTheory, 2, edges=[[0, 1]], ftype=[0]).
```

Giving $K_2^{T_1}$



The class `Flag` is written in Cython, as the performance sensitive calculations are performed here. This class contains the code that calculates slices of the multiplication table.

7.2.3 `FlagAlgebra`

The `FlagAlgebra` is a parent object, corresponding with the entire algebra $\mathcal{A}^T$. To construct a `FlagAlgebra` instance, the following is required:

- `base`: the base ring. It must contain $\mathbb{Q}$ in order to perform the calculations, and by default it is $\mathbb{Q}$,
- `theory`: the combinatorial theory, whose models form a spanning set for this algebra,
- `ftype`: the type for the elements T. If not provided, it is assumed to be the empty type.

Each `FlagAlgebraElement` must belong to one `FlagAlgebra`. While not much computation is performed inside the `FlagAlgebra` class, they provide important structural information for the SAGE framework. When a `Flag` is used in an expression, it is translated to a suitable `FlagAlgebra` parent, where the expressions can be evaluated.

7.2.4 `FlagAlgebraElement`

The elements of `FlagAlgebra`. Every object is a linear combination of a slice from the entire algebra $\mathcal{A}^T$, namely from $B^{\mathcal{H}_n^T}$. To construct an object, the following is required:

- `parent`: the parent `FlagAlgebra` object, specifying the theory, the type and the base ring,
- `n`: the size of the flags appearing in the slice this element represents,
- `values`: the coefficient for each $\mathcal{H}_n^T$ flag.

The algebra operations are implemented in `FlagAlgebraElement`. A flag F^T can be coerced into the corresponding `FlagAlgebraElement`, making it possible to use them in the calculations.

7.3 Usage

This section showcases the usage of the code. First, the standard usage is illustrated, by establishing that $\pi(K_3) \lessapprox 1/2$, that $\pi\left(K_4^{(3)}; K_5^{(3)}\right) \lessapprox 3/8$, and finally that $R(4) = 17$ using built-in theories.

Then, evaluation of various flag algebraic expressions will be showcased. The result from Chapter 6 will be derived purely from such expressions, without the use of an optimizer, and floating-point approximations/errors. Additionally, the typed expressions will be included in the SDP optimization problem, to provide further constraints in the automated proofs.

Finally, the full generality of the implementation will be illustrated through the creation of a new combinatorial theory, capturing graphs with colored vertices.

7.3.1 Simple Optimization Problems

To continue the example of Mantel's theorem, the program can approximate the value $\pi\left(K_3^{(2)}\right) = 1/2$ as follows.

```
1 #define edge and triangle flags
2 edge = GraphTheory(2, edges=[[0, 1]])
3 triangle = GraphTheory(3, edges=[[0, 1], [0, 2], [1, 2]])
4 #assert that there are no triangles
5 GraphTheory.exclude(triangle)
6 #maximize the number of edges, calculate up to size 3.
7 GraphTheory.optimize_problem(edge, 3, maximize=True)
```

```
Ftypes constructed in 0.04s
Block sizes done in 0.22s
Block sizes are [2, -3, -4]
Calculating product matrices for 1 ftypes and 3 structures
Ftype on 1 points with edges=[] is complete: : 1it [00:00, 23.92it/s]
Table calculation done in 0.05s
Target and constraint calculation done in 0.04s

CSDP 6.2.0
Iter:  0 Ap: 0.00e+00 Pobj:  0.0000000e+00 Ad: 0.00e+00 Dobj:  0.0000000e+00
Iter:  1 Ap: 1.00e+00 Pobj: -1.5891283e+01 Ad: 7.89e-01 Dobj: -1.9410590e-01
Iter:  2 Ap: 1.00e+00 Pobj: -1.3902694e+01 Ad: 9.46e-01 Dobj: -2.5686850e-01
Iter:  3 Ap: 1.00e+00 Pobj: -3.4671744e+00 Ad: 9.05e-01 Dobj: -2.7960683e-01
Iter:  4 Ap: 1.00e+00 Pobj: -6.5489281e-01 Ad: 8.45e-01 Dobj: -3.1191857e-01
Iter:  5 Ap: 1.00e+00 Pobj: -5.7185383e-01 Ad: 8.37e-01 Dobj: -4.8273150e-01
Iter:  6 Ap: 9.99e-01 Pobj: -5.0535710e-01 Ad: 9.47e-01 Dobj: -4.9478923e-01
Iter:  7 Ap: 1.00e+00 Pobj: -5.0042758e-01 Ad: 9.54e-01 Dobj: -4.9950434e-01
Iter:  8 Ap: 1.00e+00 Pobj: -5.0003767e-01 Ad: 1.00e+00 Dobj: -4.9997549e-01
Iter:  9 Ap: 1.00e+00 Pobj: -5.0000196e-01 Ad: 1.00e+00 Dobj: -4.9999928e-01
Iter: 10 Ap: 1.00e+00 Pobj: -5.0000020e-01 Ad: 1.00e+00 Dobj: -4.9999997e-01
Iter: 11 Ap: 1.00e+00 Pobj: -5.0000001e-01 Ad: 1.00e+00 Dobj: -5.0000000e-01
Iter: 12 Ap: 9.62e-01 Pobj: -5.0000000e-01 Ad: 9.60e-01 Dobj: -5.0000000e-01
Success: SDP solved
Primal objective value: -5.0000000e-01
Dual objective value: -5.0000000e-01
Relative primal infeasibility: 9.71e-16
Relative dual infeasibility: 1.54e-10
Real Relative Gap: 2.92e-10
XZ Relative Gap: 6.44e-10
DIMACS error measures: 1.02e-15 0.00e+00 3.20e-10 0.00e+00 2.92e-10 6.44e-10

0.5000000004501909
```

The code identifies automatically and displays that 1 type is relevant and displays that there are 3 non-isomorphic flags with empty type and the given size.

Then the program calculates the multiplication tables for each relevant type (here only T_1), as shown in the line `Ftype on 1 points with edges=[] is complete`. Once the multiplication table is calculated, the CSDP semidefinite solver [Bor99] finds an upper bound for the density, which is `0.5000000004501909` in this case.

The result $\pi\left(I_4^{(3)}; I_5^{(3)}\right) = \pi\left(K_4^{(3)}; K_5^{(3)}\right) = \frac{3}{8}$ can be approximated with the following program:.

```
1 ThreeGraphTheory.exclude(ThreeGraphTheory(5))
2 ThreeGraphTheory.optimize_problem(ThreeGraphTheory(4), 6, maximize=True)
```

```
...
Iter: 32 Ap: 1.00e+00 Pobj: -3.7500001e-01 Ad: 1.00e+00 Dobj: -3.7499996e-01
Iter: 33 Ap: 1.00e+00 Pobj: -3.7500000e-01 Ad: 9.24e-01 Dobj: -3.7499999e-01
Iter: 34 Ap: 9.55e-01 Pobj: -3.7500000e-01 Ad: 9.29e-01 Dobj: -3.7500000e-01
Success: SDP solved
Primal objective value: -3.7500000e-01
Dual objective value: -3.7500000e-01
Relative primal infeasibility: 8.22e-13
Relative dual infeasibility: 1.19e-09
Real Relative Gap: -8.71e-10
XZ Relative Gap: 8.89e-09
DIMACS error measures: 2.95e-12 0.00e+00 2.66e-09 0.00e+00 -8.71e-10 8.89e-09

0.3750000015557378
```

The solver can calculate with arbitrary combinatorial theory. As an example, the following shows that the Ramsey number for K_4 is $R(4) = 17$, using the method presented in [LP21]. `RamseyGraphTheory` contains two relational symbols, indicating the two edge colors.

```
1 RamseyGraphTheory.exclude(RamseyGraphTheory(4, edges=itertools.combinations(
    range(4), 2)))
2 dens = RamseyGraphTheory.optimize_problem(RamseyGraphTheory(2), 6, maximize=
    False)
3 1/dens
```

```
...
```

```
Iter: 25 Ap: 1.00e+00 Pobj:  5.8823370e-02 Ad: 9.49e-01 Dobj:  5.8822728e-02
Iter: 26 Ap: 1.00e+00 Pobj:  5.8823517e-02 Ad: 1.00e+00 Dobj:  5.8823461e-02
Iter: 27 Ap: 9.54e-01 Pobj:  5.8823529e-02 Ad: 9.55e-01 Dobj:  5.8823525e-02
Success: SDP solved
Primal objective value: 5.8823529e-02
Dual objective value: 5.8823525e-02
Relative primal infeasibility: 1.26e-12
Relative dual infeasibility: 1.68e-10
Real Relative Gap: -3.57e-09
XZ Relative Gap: 1.09e-09
DIMACS error measures: 1.80e-12 0.00e+00 6.97e-10 0.00e+00 -3.57e-09 1.09e-09

17.000001320374114
```

7.3.2 Evaluating Expressions

On top of the above optimizer, the software can be used to interactively evaluate flag algebraic expressions. The flags, as part of the flag algebra, can be added and multiplied, as one would expect.

```
1 G = GraphTheory
2 cherry = G(3, edges=[[0, 1], [1, 2]])
3 empty_3 = G(3)
4 empty_2 = G(2)
5 empty_3 + cherry, empty_2*empty_2
```

```
(Flag Algebra Element over Rational Field
1 - Flag on 3 points, ftype from [] with edges=[]
0 - Flag on 3 points, ftype from [] with edges=[[0, 2]]
1 - Flag on 3 points, ftype from [] with edges=[[0, 2], [1, 2]]
0 - Flag on 3 points, ftype from [] with edges=[[0, 1], [0, 2], [1, 2]],
Flag Algebra Element over Rational Field
1   - Flag on 4 points, ftype from [] with edges=[]
2/3 - Flag on 4 points, ftype from [] with edges=[[0, 3]]
1/3 - Flag on 4 points, ftype from [] with edges=[[0, 3], [1, 3]]
0   - Flag on 4 points, ftype from [] with edges=[[0, 3], [1, 3], [2, 3]]
```

```
2/3 - Flag on 4 points, ftype from [] with edges=[[0, 2], [1, 3]]
1/3 - Flag on 4 points, ftype from [] with edges=[[0, 2], [0, 3], [1, 3]]
0   - Flag on 4 points, ftype from [] with edges=[[0, 2], [0, 3], [2, 3]]
0   - Flag on 4 points, ftype from [] with edges=[[0, 2], [0, 3], [1, 3], [2,
    3]]
1/3 - Flag on 4 points, ftype from [] with edges=[[0, 2], [0, 3], [1, 2], [1,
    3]]
0   - Flag on 4 points, ftype from [] with edges=[[0, 2], [0, 3], [1, 2], [1,
    3], [2, 3]]
0   - Flag on 4 points, ftype from [] with edges=[[0, 1], [0, 2], [0, 3], [1,
    2], [1, 3], [2, 3]])
```

The Python syntax allows the renaming of theories, for simplicity `G = GraphTheory` from now on. Any `FlagAlgebraElement` can be written with a larger size. To increase the size where an element is written, the bit shift operator is overloaded.

```
edge = G(2, edges=[[0, 1]])
edge << 1
```

```
Flag Algebra Element over Rational Field
0   - Flag on 3 points, ftype from [] with edges=[]
1/3 - Flag on 3 points, ftype from [] with edges=[[0, 2]]
2/3 - Flag on 3 points, ftype from [] with edges=[[0, 2], [1, 2]]
1   - Flag on 3 points, ftype from [] with edges=[[0, 1], [0, 2], [1, 2]]
```

The optimizer runs on any target expression. The following example minimizes $K_2 + I_3/2$ (the optimum is attained at the complement of a balanced 4-partite construction, with value $7/16 = 0.4375$).

```
G.optimize_problem(edge + empty_3/2, 3, maximize=False)
```

```
...
Result is 0.43749999801640244
```

The optional `ftype` parameter in the definition of `Flag` can be used to indicate the type of a flag. The calculations can be performed with typed expressions, resulting in a `FlagAlgebraElement` with the matching type. The only exception is the averaging operator, it returns a `FlagAlgebraElement`

with a smaller type. The `Flag.project()` corresponds to the calculation performed in 2.5.

```
pointed_edge = G(2, edges=[[0, 1]], ftype=[0])
pointed_cherry = G(3, edges=[[0, 1], [1, 2]], ftype=[1])
pointed_edge + pointed_cherry/2, pointed_edge*pointed_edge, pointed_cherry.
    project()
```

```
(Flag Algebra Element over Rational Field
0   - Flag on 3 points, ftype from [0] with edges=[]
1/2 - Flag on 3 points, ftype from [0] with edges=[[0, 2]]
0   - Flag on 3 points, ftype from [1] with edges=[[0, 2]]
1/2 - Flag on 3 points, ftype from [0] with edges=[[0, 2], [1, 2]]
3/2 - Flag on 3 points, ftype from [2] with edges=[[0, 2], [1, 2]]
1   - Flag on 3 points, ftype from [0] with edges=[[0, 1], [0, 2], [1, 2]],
Flag Algebra Element over Rational Field
0 - Flag on 3 points, ftype from [0] with edges=[]
0 - Flag on 3 points, ftype from [0] with edges=[[0, 2]]
0 - Flag on 3 points, ftype from [1] with edges=[[0, 2]]
0 - Flag on 3 points, ftype from [0] with edges=[[0, 2], [1, 2]]
1 - Flag on 3 points, ftype from [2] with edges=[[0, 2], [1, 2]]
1 - Flag on 3 points, ftype from [0] with edges=[[0, 1], [0, 2], [1, 2]],
Flag Algebra Element over Rational Field
0   - Flag on 3 points, ftype from [] with edges=[]
0   - Flag on 3 points, ftype from [] with edges=[[0, 2]]
1/3 - Flag on 3 points, ftype from [] with edges=[[0, 2], [1, 2]]
0   - Flag on 3 points, ftype from [] with edges=[[0, 1], [0, 2], [1, 2]])
```

The optimizer allows the addition of density constraints. They can be added, by translating them to a positivity constraint. For example the expression $H_1+H_2 \leq 1/2+H_3$ translates to $1/2-H_1-H_2+H_3 \geq 0$, and then to the linear constraint $\sum_F \Phi(F)d(1/2-H_1-H_2+H_3, F) \geq 0$ (with the notation from Section 7.1), which is then added to the optimization problem. The example below maximizes the number of triangles, assuming that the density of edges is at most $1/2$ (which translates to the positivity assumption $1/2 - K_2 \geq 0$).

```
G.optimize_problem(triangle, 4, maximize=True, positives=[1/2 - edge])
```

```
...
Result is 0.3535533922528127
```

Note $1/\sqrt{8} \approx 0.3535533$, which is attained when $1/\sqrt{2}$ fraction of the points span a complete graph, while the rest has no edges.

Constraints can involve flags with non-empty types. The program translates that to an assumption satisfied for all matching types in the model. For example, instead of assuming that the density of edges is at most 1/2, it is possible to specify that each vertex has degree at most 1/2 with the constraint $1/2 - K_2^{T_1} \geq 0$.

```
G.optimize_problem(triangle, 4, maximize=True, positives=[1/2 - pointed_edge])
```

```
...
Result is 0.25000000071422
```

The optimum is attained with two cliques, each having 1/2 density.

As the `FlagAlgebra` can be defined over any ring containing the rationals, it is possible to evaluate expressions involving variables. The following example involves the base polynomial ring $\mathbb{Q}[x]$.

```
R.<x> = PolynomialRing(QQ)
(x + pointed_cherry) * (pointed_edge - 1/2 + x)
```

```
Flag Algebra Element over Univariate Polynomial Ring in x over Rational Field
x^2 - 1/2*x       - Flag on 4 points, ftype from [0] with edges=[]
x^2 - 1/6*x       - Flag on 4 points, ftype from [0] with edges=[[0, 3]]
x^2 - 1/2*x       - Flag on 4 points, ftype from [1] with edges=[[0, 3]]
x^2 - 1/6*x       - Flag on 4 points, ftype from [0] with edges=[[0, 3], [1,
    3]]
x^2 - 1/2*x       - Flag on 4 points, ftype from [2] with edges=[[0, 3], [1,
    3]]
x^2 + 1/2*x - 1/6 - Flag on 4 points, ftype from [3] with edges=[[0, 3], [1,
    3]]
x^2 - 1/6*x       - Flag on 4 points, ftype from [0] with edges=[[0, 3], [1,
    3], [2, 3]]
x^2 + 3/2*x + 1/2 - Flag on 4 points, ftype from [3] with edges=[[0, 3], [1,
    3], [2, 3]]
```

```
x^2 - 1/6*x       - Flag on 4 points, ftype from [0] with edges=[[0, 2], [1,
    3]]
x^2 + 1/2*x - 1/6 - Flag on 4 points, ftype from [0] with edges=[[0, 2], [0,
    3], [1, 3]]
x^2 - 1/6*x       - Flag on 4 points, ftype from [1] with edges=[[0, 2], [0,
    3], [1, 3]]
x^2 + 1/6*x       - Flag on 4 points, ftype from [0] with edges=[[0, 2], [0,
    3], [2, 3]]
x^2 - 1/2*x       - Flag on 4 points, ftype from [1] with edges=[[0, 2], [0,
    3], [2, 3]]
x^2 + 1/6*x       - Flag on 4 points, ftype from [0] with edges=[[0, 2], [0,
    3], [1, 3], [2, 3]]
x^2 - 1/6*x       - Flag on 4 points, ftype from [1] with edges=[[0, 2], [0,
    3], [1, 3], [2, 3]]
x^2 + 7/6*x + 1/3 - Flag on 4 points, ftype from [3] with edges=[[0, 2], [0,
    3], [1, 3], [2, 3]]
x^2 + 1/2*x - 1/6 - Flag on 4 points, ftype from [0] with edges=[[0, 2], [0,
    3], [1, 2], [1, 3]]
x^2 + 1/6*x       - Flag on 4 points, ftype from [0] with edges=[[0, 2], [0,
    3], [1, 2], [1, 3], [2, 3]]
x^2 + 5/6*x + 1/6 - Flag on 4 points, ftype from [2] with edges=[[0, 2], [0,
    3], [1, 2], [1, 3], [2, 3]]
x^2 + 1/2*x       - Flag on 4 points, ftype from [0] with edges=[[0, 1], [0,
    2], [0, 3], [1, 2], [1, 3], [2, 3]]
```

7.3.3 Explicit Squares

The CSDP library uses a primal-dual interior-point method to find the optimal solution in the semidefinite cone. While it provides valuable information about the optimal value, often it lacks insight on why that result is optimal, since the solution lies inside an often over-parametrized space. As an example, Mantel's triangle problem is not over-parametrized, therefore the certificates are tight and exact.

```
1 GraphTheory.exclude(GraphTheory(3))
```

```python
solution = GraphTheory.optimize_problem(GraphTheory(2), 3, maximize=True,
    certificate=True)
solution[2]['X'][0]
```

```
...
[[0.49992307032862443, -0.4999615293760331],
[-0.4999615293760331, 0.4999999976849489]]
```

This shows that the positive semidefinite matrix

$$M = \begin{pmatrix} 1/2 & -1/2 \\ -1/2 & 1/2 \end{pmatrix}$$

multiplied with the vector of flags $v = \left(\overset{\bullet}{\circ}, \ \overset{\bullet}{\underset{\circ}{|}} \right) = \mathcal{H}_2^{E_1}$ provides the bound

$$0 \leq vMv^T = \frac{1}{2} \left(\overset{\bullet}{\circ} - \overset{\bullet}{\underset{\circ}{|}} \right)^2$$

which after averaging, becomes

$$0 \leq \left[\!\left[vMv^T \right]\!\right]_{E_1} = \frac{1}{2} \, {}_{\circ}\overset{\bullet}{\ }{}_{\circ} - \frac{1}{6} \, \overset{\bullet}{\circ\!\!-\!\!\circ} + \frac{1}{6} \, {}_{\circ}\!\overset{\bullet}{\wedge}\!{}_{\circ} \leq \frac{1}{2} - \overset{\bullet}{\underset{\circ}{|}}$$

proving that $\pi(K_3) \leq 1/2$. On the other hand, the larger $\pi(K_4)$ problem provides the following matrices

```python
GraphTheory.exclude(GraphTheory(4))
solution = GraphTheory.optimize_problem(GraphTheory(2), 4, maximize=True,
    certificate=True)
print(matrix(solution[2]['X'][0]), "\n\n", matrix(solution[2]['X'][1]))
```

```
...
[   1.166661022199803   -0.5833318705364714   -0.5833318704531831
    -0.5892459845599405]
[  -0.5833318705364714   0.6666664651776371  -0.08333323295064955
    0.2946243708832331]
[  -0.5833318704531831  -0.08333323295064955   0.6666664650943488
    0.29462437084455967]
```

```
[ -0.5892459845599405   0.2946243708832331  0.29462437084455967
   0.8287230904732283]


[   0.1666664674317131   0.02295563377727656  0.022955633777276493
   -0.3333331300171468]
[  0.02295563377727656    0.6136564015308313   -0.3535719465199399
   -0.045910849434157616]
[ 0.022955633777276493   -0.3535719465199399    0.6136564015308311
   -0.045910849434157436]
[  -0.3333331300171468 -0.045910849434157616 -0.045910849434157436
   0.6666666595244639]
```

which gives less insight to what an exact solution could look like. The key observation was that the squares in these certificates can be guessed, and with a proper guess the optimization problem is not over-parametrized. Following the notation from Chapter 6, the square in Mantel's theorem is simply $\left(K_2^{E_1} - \frac{1}{2}E_1\right)^2$ before scaling. The problem $\pi(K_4)$ has an exact certificate utilizing only the squares $\left(K_2^{E_1} - \frac{2}{3}E_1\right)^2$ and $\left(K_3^{E_2} - \frac{1}{3}E_2\right)^2$. The generalization of this observation is the main content of Chapter 6.

In a similar setting, as outlined in Section 6.6, the proof of $\pi\left(K_4^{(3)}; K_5^{(3)}\right) \leq 3/8$ follows from the squares (using the same notation)

$$\left(E_3^{P_1} - \frac{3}{4}P_1\right)^2,$$
$$\left(L_a^{P_2} - L_b^{P_2}\right)^2,$$
$$\left(M_a^{P_3} + M_b^{P_3} + M_c^{P_3} - \frac{1}{2}P_3\right)^2,$$
$$\left(E_4^{P_3} - \frac{1}{2}P_3\right)^2,$$
$$\left(N^{Q_4} - \frac{1}{2}Q_4\right)^2,$$
$$\left(O_a^{T_4} - O_b^{T_4}\right)^2.$$

While the above list involves simple expressions, the linear combination of these squares still over-parametrizes the region where the proof is exact. SAGE has a way to construct a

polytope from the constraints and find the simple rational coefficients for the squares displayed in Section 6.5. This was performed with the following calculation.

```
em5 = ThreeGraphTheory(5)
ThreeGraphTheory.exclude(em5)

em3p1 = ThreeGraphTheory(3, ftype_points=[0])
sq1 = (em3p1 - 3/4).mul_project(em3p1 - 3/4)

la4p2 = ThreeGraphTheory(4, ftype_points=[0, 1], edges=[[0, 2, 3]])
lb4p2 = ThreeGraphTheory(4, ftype_points=[0, 1], edges=[[1, 2, 3]])
sq2 = (la4p2 - lb4p2).mul_project(la4p2 - lb4p2)

ma4p3 = ThreeGraphTheory(4, ftype_points=[0, 1, 2], edges=[[0, 1, 3]])
mb4p3 = ThreeGraphTheory(4, ftype_points=[0, 1, 2], edges=[[0, 2, 3]])
mc4p3 = ThreeGraphTheory(4, ftype_points=[0, 1, 2], edges=[[1, 2, 3]])
sq3 = (ma4p3 + mb4p3 + mc4p3 - 1/2).mul_project(ma4p3 + mb4p3 + mc4p3 - 1/2)

em4p3 = ThreeGraphTheory(4, ftype_points=[0, 1, 2])
sq4 = (em4p3 - 1/2).mul_project(em4p3 - 1/2)

n5q4 = ThreeGraphTheory(5, ftype_points=[0, 1, 2, 3], edges=[[0, 1, 2]])
sq5 = (n5q4 - 1/2).mul_project(n5q4 - 1/2)

oa5t4 = ThreeGraphTheory(5, ftype_points=[0, 1, 2, 3], edges=[[0, 1, 4]])
ob5t4 = ThreeGraphTheory(5, ftype_points=[0, 1, 2, 3], edges=[[2, 3, 4]])
sq6 = (oa5t4 - ob5t4).mul_project(oa5t4 - ob5t4)

M = Matrix([
(sq1<<(6-sq1.size())).values(),
(sq2<<(6-sq2.size())).values(),
(sq3<<(6-sq3.size())).values(),
(sq4<<(6-sq4.size())).values(),
(sq5<<(6-sq5.size())).values(),
(sq6<<(6-sq6.size())).values()
```

```
33 ])
34
35 optimal_construction_densities = matrix(QQ, 1, 2102, {(0, 0): 5/16, (0, 4):
       15/32, (0, 875): 3/16, (0, 2101): 1/32})
36 target = (ThreeGraphTheory(4, edges=[])<<2).values()
37 opt = vector([3/8]*2102) - target
38
39 #from duality, the hypergraphs appearing in the optimal construction
40 equality_ids = [0, 4, 875, 2101]
41
42 #equations forming the faces of the polyhedon
43 ieqs = [[opt[ii]]+[-M[jj, ii] for jj in range(6)] for ii in range(2102) if (ii
       not in equality_ids)] \
44 + [[0]*(6-ii)+[1]+[0]*ii for ii in range(6)]
45 eqns = [[opt[ii]]+[-M[jj, ii] for jj in range(6)] for ii in equality_ids]
46
47 solspace = Polyhedron(ieqs=ieqs, eqns=eqns, base_ring=QQ, backend='cdd')
48 print(solspace, "\n", "\n".join(map(str, solspace.vertices())))
```

```
A 3-dimensional polyhedron in QQ^6 defined as the convex hull of 6 vertices
A vertex at (2/3, 143/837, 928/837, 676/837, 1856/837, 464/837)
A vertex at (2/3, 7/45, 32/27, 28/27, 64/45, 16/45)
A vertex at (2/3, 4/27, 32/27, 28/27, 40/27, 10/27)
A vertex at (2/3, 4/27, 212/189, 160/189, 424/189, 106/189)
A vertex at (2/3, 7/45, 52/45, 16/15, 64/45, 16/45)
A vertex at (2/3, 37/207, 196/207, 56/69, 544/207, 136/207)
```

The coefficients from Equation (6.3) were selected to fall in this polyhedron.

7.4 Constructing New Theories

As outlined in 7.2, the construction of a `CombinatorialTheory` requires a generator and an identifier. By creating a theory for graph with colored vertices, this section showcases the steps to create a new theory. Furthermore, it highlights that the rest of the algorithm can be used on this newly created theory, without any additional work.

```
def identifier_cgt(n, ftype_points, edges, colorA, colorB):
  g_parts = [[ii] for ii in ftype_points] + \
    [[ii for ii in range(n) if ii not in ftype_points]] + \
    [[n], [n+1]]
  g_verts = list(range(n+2))
  g_edges = list(edges) + [(xx[0], n) for xx in colorA] + [(xx[0], n+1) for xx
     in colorB]
  g = Graph([g_verts, g_edges], format='vertices_and_edges')
  blocks = tuple(g.canonical_label(partition=g_parts).edges(labels=None, sort=
    True))
  ftype_points = tuple(range(len(ftype_points)))
  return (n, ftype_points, blocks)


def generator_cgt(n):
  for xx in GraphTheory.generate_flags(n):
    unique = []
    edges = xx.blocks()['edges']

    for yy in itertools.product(range(2), repeat=int(n)):
      yy = list(yy)
      colorA = [[ii] for ii, oo in enumerate(yy) if oo==0]
      colorB = [[ii] for ii, oo in enumerate(yy) if oo==1]
      iden = identifier_cgt(n, [], edges, colorA, colorB)
      if iden not in unique:
        unique.append(iden)
        yield {"edges":edges, "colorA":colorA, "colorB":colorB}

ColGraphT = CombinatorialTheory('2ColoredGraphTheory', generator_cgt,
    identifier_cgt, edges=2, colorA=1, colorB=1)
```

The creation of a theory starts with a fixed signature. In this case `edges` will be a binary relation and `colorA`, `colorB` are unary relations, indicating the color classes. The code above defines two functions `identifier_cgt` and `generator_cgt`.

1. `identifier_cgt`, when a model (with ftype) is received as an input, it returns a unique identifier for that model, such that isomorphic models return equal identifiers. The code uses graph canonical labelings as a black-box for this purpose [MP14, JK07]. The graph is extended with two additional vertices, each connected to a given color class of the vertices. The canonical labeling is then restricted to keep these vertices in place.

2. `generator_cgt`, when an integer n is received as an input, it returns the possible non-isomorphic models with that size. This example code enumerates the non-isomorphic graphs on n vertices (using the corresponding code from `GraphTheory`), then tries all possible colorings, checking with the `identifier_cgt` for unique elements. The models must be returned as a dictionary, one entry for each signature element.

After the creation of a new theory, the functionality from previous chapters is available. The code below maximizes the number of cherries, where one cherry has color B, the rest is A colored.

```
aab_cherry = ColGraphT(3, edges=[[0, 1], [1, 2]], colorA=[[0], [1]], colorB
    =[[2]])
ColGraphT.optimize_problem(aab_cherry, 3, maximize=True)
```

```
...
Result is 0.2222222383751336
```

Note that the optimum is attained at the balanced blowup of the target colored cherry.

Chapter 8

Inductive Satisfiability Coding Lemma

8.1 Introduction

This chapter explores a combinatorial approach to generalize the satisfiability coding lemma [PPZ97] in order to prove size lower-bounds for bounded-depth circuits computing PAR.

In proving lower bounds against small-depth Boolean circuits, there are two main approaches exploiting the bounded-depth property: bottom-up and top-down. Bottom-up methods, with random restrictions and switching lemmas as typical examples [FSS81, Ajt83, Hå86, Ros08, Hå14, Ros18], study circuits incrementally, starting from the input gates and moving up, towards the output, usually simplifying the circuit and reducing the depth. In contrast, top-down methods analyze the circuit from the output gate downwards. By exploiting the structure of the function, top-down approaches try to find a mistake in the computation. Top-down methods have successfully proven exponential lower bounds for monotone circuits and formulas [KW90, RW92, GP18, PR17, GGKS20], and exponential lower bounds for depth-3 circuits [HJP95, PPZ97, PSZ00, IPZ01, PPSZ05, MW19, ST18, FGT22, GGM23], and very recently, an exponential lower bound for depth-4 circuits [GRSS24].

The satisfiability coding lemma, a notable example of a top-down method, provides a

tight and insightful exponential lower bound for depth-3 circuits computing PAR [PPZ97]. Despite these successes, generalizing the satisfiability coding lemma to higher-depth circuits, matching the generality of random restrictions, remains an important and challenging open problem [HJP95, PPZ97, MW19, GRSS24].

This chapter attempts to address this challenge by introducing the notion of "cones", a generalization of critical clauses. Cones provide a combinatorial structure allowing the usage of a satisfiability coding lemma-style arguments in any function, thus offering a step towards extending the satisfiability coding lemma to higher depths. However, an inherent difficulty remains: cones in a given layer of a circuit are only partially related to cones in the next layer, causing the induction to fail in general. To bridge this gap, a new structural condition "nice" will be introduced, which informally states that the number of cones is the same as the number of satisfying assignments (or unsatisfying assignments, depending on $\wedge$ or $\vee$ gates). A more precise definition is available in Section 8.3.2. This condition is satisfied by a broad class of circuits, including known optimal constructions, although many examples of circuits exist that violate this structural condition.

8.1.1 Outline of the Chapter

For convenience, the next Section 8.2 recalls the known "divide and conquer" construction for PAR, with varying divisions. In particular, the following construction will be provided.

Theorem 21. *Given a factoring of n as $n = n_1 n_2 ... n_{d-1}$, there exists a circuit computing* PAR *on n input bits having depth d and sizes $|V_i| = \frac{n}{n_1 n_2 ... n_i} 2^{n_i}$.*

Section 8.3 includes additional terminology. In particular, the associated sensitive edges in Section 8.3.1 and the associated cones Section 8.3.2 will be defined there in a top-down manner. Using those definitions, nice circuits will be defined, which is the structural assumption required to make the inductions work. Section 8.4 includes the proof of the main Corollary 66 and Lemma 65, which provides a relation between the number of cones in two consecutive layers of a nice circuit. This relation will be inductively applied to connect the number of cones in the output and input gates of a circuit in the main result of the chapter, which is

Theorem 64. Section 8.4.1 includes the application of the main Theorem 64 to PAR, giving the following.

Theorem 22. *Suppose a depth d, circuit $\mathcal{C}$ computes* PAR *on n bits, with each gate in $\mathcal{C}$ nice. Find a (not necessarily integer) sequence $n_1, n_2, ..., n_{d-1}$ such that $|V_i(\mathcal{C})| = \frac{n}{n_1 n_2 ... n_i} 2^{n_i - 1}$ for all $i \le d$. If each $n_i \ge 2$, then it must hold that $\prod_i n_i \ge n$.*

Note that under the niceness condition, the construction for PAR is optimal. The proof of the technical Lemma 65 is included in Section 8.4.2. Section 8.5 concludes the chapter with closing thoughts and open questions.

8.2 Small Circuits computing Parity

This section showcases the "divide and conquer" based construction. The input is split into smaller blocks, and each layer computes a parity-like function on larger and larger groups of blocks. In particular, if the inputs are split into $n = n_1 n_2 ... n_{d-1}$ parts, then at layer i, each gate computes n_i many combinations of parity functions, where each parity is computed on $n_1 n_2 ... n_{i-1}$ inputs. The precise size bound following from this construction is as follows.

Theorem 21. *Given a factoring of n as $n = n_1 n_2 ... n_{d-1}$, there exists a circuit computing* PAR *on n input bits having depth d and sizes $|V_i| = \frac{n}{n_1 n_2 ... n_i} 2^{n_i}$.*

Note that the construction can be more generally stated with the extended sequence $n = n_0 n_1 ... n_{d-1} n_d$, where $n_0 = 1, n_d = 1$. The 0th layer size constraint translates to $|V_0| = 2n$, matching the number of the n input bits and their negations. Similarly, the dth layer size constraint, regarding the output gates, gives $|V_d| = 2$, matching the setting where the multi output function $X \mapsto \{\text{PAR}(X), \neg\,\text{PAR}(X)\}$ is computed.

Proof. Work in the extended indexing, where $n_0 = n_d = 1$ is included. For short, use p_i for the product of the first i elements $p_i := n_0 n_1 ... n_i$. Suppose inductively there is a circuit C^i on p_i input bits, with two outputs computing ($\text{PAR}, \neg\,\text{PAR}$) using $i+1$ depth and $\left|V_j(C^i)\right| = \frac{p_i}{p_j} 2^{n_j}$ gates in the layers. This is trivially true for $i = 0$, the input gates can be directly used as output gates, giving $|V_0(C^0)| = 2$.

By listing the satisfying (or unsatisfying) assignments, there is a simple CNF (or DNF) on m input bits, computing $X \mapsto \{\mathrm{PAR}(X), \neg\,\mathrm{PAR}(X)\}$ with 2^m clauses.

To finish the induction, partition the p_{i+1} inputs into blocks of size p_i, compute C^i on each block. Due to duality, it can be assumed that the top gate of C^i is OR. Take the depth 2 CNF that computes $X \mapsto \{\mathrm{PAR}(X), \neg\,\mathrm{PAR}(X)\}$ on the $2n_{i+1}$ outputs of each C^i, corresponding with the parity in each block and their negation. Since the consecutive OR layers can be merged together, this provides a depth $i+2$ circuit. Note further that $\left|V_{i+1}(C^{i+1})\right| = 2^{n_{i+1}}$ and for all smaller $j < i+1$ the size of the layer is $\left|V_j(C^{i+1})\right| = n_{i+1}\left|V_j(C^i)\right| = \frac{p_{i+1}}{p_j}2^{n_j}$ as required. □

The resulting circuit's gates can be defined explicitly. The gates of V_i can be indexed by the set $\left[n/p_i\right] \times \{0,1\}^{[n_i]}$. In particular the gates are $v^i_{j,S} \in V_i$ such that $j \in \left[n/p_i\right]$ and $S : [n_i] \longrightarrow \{0,1\}$.

The 0th layer contains the input gates and their negations. $v^0_{j,S}$ is the input gate j if $S(0) = 0$, otherwise when $S(0) = 1$, then $v^0_{j,S}$ is the negation of the input gate j.

For the following layers, when $\mathcal{L}(V_i) = \wedge$, then gate $v^i_{j,S}$ computes

$$v^i_{j,S} := \bigwedge_{\substack{l\in[n_i]\\ T\subseteq[n_{i-1}]\\ \mathrm{PAR}(T)=S(l)}} v^{i-1}_{jn_i+l,T}, \tag{8.1}$$

and dually when $\mathcal{L}(V_i) = \vee$ then

$$v^i_{j,S} := \bigvee_{\substack{l\in[n_i]\\ T\subseteq[n_{i-1}]\\ \mathrm{PAR}(T)=S(l)}} v^{i-1}_{jn_i+l,T}.$$

When $\mathcal{L}(V_i) = \wedge$, the gates at layer i compute

$$v^i_{j,S} = \bigwedge_{l\in S} \mathrm{PAR}\left(B_l\right) \bigwedge_{l\notin S} \neg\,\mathrm{PAR}\left(B_l\right),$$

and dually when $\mathcal{L}(V_i) = \vee$ then

$$v_{j,S}^{i} = \bigvee_{l \in S} \mathrm{PAR}\left(B_l\right) \bigvee_{l \notin S} \neg \mathrm{PAR}\left(B_l\right).$$

Here B_l is just a partitioning of a p_{i+1} sized subset of the input bits into p_i sized parts. So a typical gate in this circuit is computing an AND or OR of smaller parity functions.

Note that, assuming d is constant, and $n \to \infty$, the smallest total size is attained when each V_i layer has a similar size. The resulting construction, with

$$n_i \approx n^{\frac{1}{d-1}} + \frac{2i - d}{2d - 2} \log_2 n$$

gives the total size

$$\approx 2^{n^{\frac{1}{d-1}}} n^{\frac{d-2}{2d-2}}.$$

Due to the inexact nature of this optimum, the rest of the chapter will only concern the $n_1 n_2 ... n_{d-1}$ sequence, and the relationship between them compared to the size of each layer. Namely, that this construction provides $|V_i| = \frac{n}{p_i} 2^{n_i}$.

8.3 Additional Notation

The symmetric difference of two sets here is denoted by Δ. For later definitions, an ordering of the input bits is needed. The result holds with any fixed ordering, for simplicity the rest of the result will use the natural ordering coming from $S = [n] = \{0, 1, ..., n-1\}$. Correspondingly, for sets $A_1, A_2 \subseteq [n]$, the lexicographic ordering is defined as $A_1 \leq A_2$, if $\sum_{i \in A_1} 2^i \leq \sum_{i \in A_2} 2^i$.

There is a natural way to extend $\{0, 1\}^S$ to a graph representing sensitivity, where for $X, Y \subseteq S$ the pair XY is an edge if they differ in exactly one element. Write

$$G(S) := \left\{ XY : |X \Delta Y| = 1 \right\}$$

to represent the hypercube graph on vertex set $V(G(S)) = \{0, 1\}^S$.

8.3.1 Sensitive Edges

A function f has $A(f) = \{XY \in G([n]) : f(X) \neq f(Y)\}$ sensitive edges. They are pairs of inputs differing in one bit, such that the output of the function changes. Note that normalizing by 2^{n-1}, the size of $A(f)$ agrees with the average sensitivity appearing in the literature. Given a circuit $\mathcal{C}$, define the associated sensitive edges $A : V(\mathcal{C}) \longrightarrow \{0,1\}^{G(S)}$ recursively from the outputs in a top-down way for each gate.

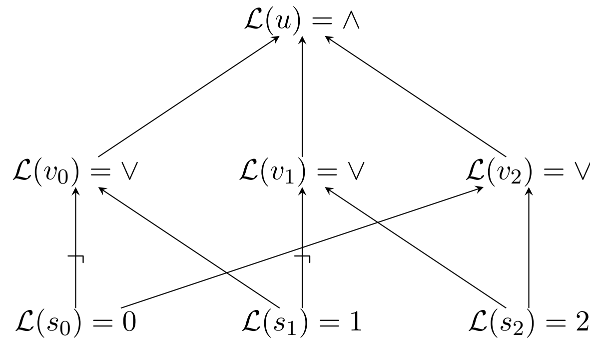
- When $XY \in G(S)$ and $f_{\mathcal{C}}(X) \neq f_{\mathcal{C}}(Y)$, there must be at least one $v \in \mathcal{O}$ in the output, such that $f_v(X) \neq f_v(Y)$. Pick exactly one such v and add XY to $A(v)$. This ensures they are disjoint and contain every element from $A(\mathcal{C})$ exactly once.
- Suppose $A(v)$ is defined for all $v \in V_{i+1}(\mathcal{C})$, thus for every $XY \in A(\mathcal{C})$ there is a corresponding $v \in V_{i+1}(\mathcal{C})$ with $XY \in A(v)$. For the same XY edge, there must be at least one $u \in V_i(\mathcal{C})$, such that $f_u(X) = f_v(X)$ and $f_u(Y) = f_v(Y)$ with $\overline{uv} \in E(\mathcal{C})$. Pick exactly one such u and add XY to $A(u)$.

This total sensitive edge set is preserved over different layers of the circuit using this construction, and inside each layer, the sensitive edge sets remain disjoint. For each layer $0 < i \leq d(\mathcal{C})$ it holds that

$$\bigsqcup_{v \in V_i(\mathcal{C})} A(v) = A(\mathcal{C}).$$

Note that this construction is not canonical, there could be many possible choices. Additionally, $A(v) \subseteq A(f_v)$, not necessarily equal, due to, for example, the exclusion of repetitions.

Example 58. *As an example, consider the following circuit.*



For short, concatenate the elements of a set, so for example abc represents the set $\{a, b, c\}$*. Using this notation, this circuit has satisfying assignments* $012, 12, 2$*. Since there is only one output* u*, at layer* 2*, all the sensitive edges are associated to* u*,*

$$A(u) = \{(012, 02), (2, 02), (012, 01), (12, 1), (2, \emptyset)\}.$$

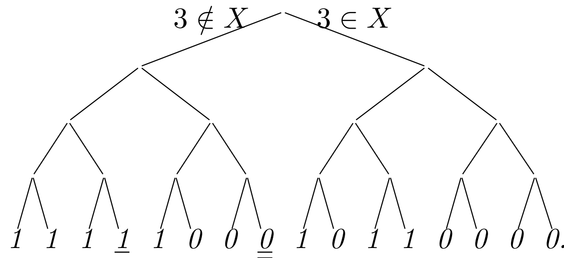
For layer 1*, when determining the associated sensitive edges, note that there is a choice for the edge* $(12, 1)$*. Both* v_1 *and* v_2 *output a different value on the pair* $(12, 1)$*, so it could be assigned to any one of them. Assuming it is assigned to* v_1*, a possible way is to have the associated sensitive edges on layer* 1 *as:* $A(v_0) = \{(012, 02), (2, 02)\}, A(v_1) = \{(12, 1), (012, 01)\}$ *and* $A(v_2) = \{(2, \emptyset)\}$*.*

Remark 59. *In the block construction for PAR mentioned in Section 8.2, the associated edge sets are unique, there is no choice at any gate.*

8.3.2 Cones

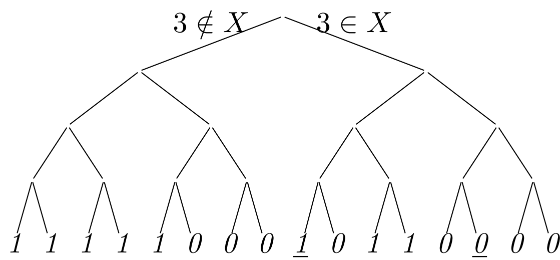
For the definition of cones the fixed ordering of the input bits will be used, in particular, the induced lexicographic ordering between subsets. Given a non-constant function $f : \{0, 1\}^{[n]} \longrightarrow \{0, 1\}$ and any $X \subseteq [n]$, write $D_f(X)$ for the lexicographically smallest D such that $f(X) \neq f(X \Delta D)$. A cone is any pair $(X, D_f(X))$ where $|D_f(X)| = 1$. A cone is a e-cone if $f(X) = e$ where $e \in \{0, 1\}$.

Example 60. *Suppose a function* f *has* $[n] = \{0, 1, 2, 3\}$ *input bits, and is true exactly on the values* $\emptyset, \{0\}, \{1\}, \{0, 1\}, \{2\}, \{3\}, \{1, 3\}, \{0, 1, 3\}$*. Then the decision tree of* f*, when enumerating according to the reversed ordering based on* n*, looks like*



Where the first split decides if 3 is included, the second split decides if 2 is included and so on. Note that the single underlined satisfying assignment is $X = \{0,1\}$ *and the lexicographically closest unsatisfying assignment is the double underlined input* $Y = \{0,1,2\}$ *with difference* $D_f(X) = \{2\}$. *Note that they differ in one element, hence* $(\{0,2\},\{2\})$ *is a 1-cone.*

As a negative example, on the same function, take the unsatisfying assignment $Y = \{0,2,3\}$ *(similarly indicated with a single underline in the illustration below). The lexicographically closest satisfying assignment is* $\{3\}$ *(double underlined), with difference* $D_f(Y) = \{0,2\}$.



Hence, the pair $(\{0,2,3\},\{0,2\})$ *is not a 0-cone.*

Remark 61.

- *The name "cone" comes from the illustration in Example 60. If* $(X, D_f(X))$ *is a 1-cone, then for all the lexicographically smaller* $D < D_f(X)$*, the values* $X \Delta D$ *are all satisfying, meaning that every path returns a satisfying assignment, forming a "cone" shape of 1 values.*

- *Cones are a generalization of the critical clauses from [PPZ97]. For example, if* $(X, D_f(X))$ *is a 1-cone for* f*, using the Example 60, if one follows the decision path towards the unsatisfying assignment* $Y = X \Delta D_f(X)$*, at the decision step* $D_f(X)$*, choosing the direction towards* X *would result in only satisfying assignments, so in order to get an unsatisfying assignment the choice at* $D_f(X)$ *is forced.*

0-cones will help encode satisfying assignments in $\wedge$ gates and 1-cones will help encode unsatisfying assignments in $\vee$ gates. They are also the key quantity in the inductive proof.

In order to not double count them, the associated edges will help distinguish between them. Given $v \in V(\mathcal{C})$ and $e \in \{0,1\}$, write

$$C_e(v) := \big\{(X,D) : (X,D) \text{ is an } e\text{-cone for } f_v, \{X, X\Delta D\} \in A(v)\big\}$$

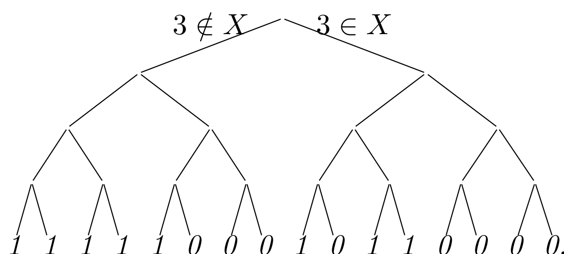
for the e-cones in f_v whose difference agrees with an associated edge. When $\mathcal{L}(v) = \wedge$, use $C(v) = C_1(v)$. Dually, when $\mathcal{L}(v) = \vee$, use $C(v) = C_0(v)$. Similarly define $C(v)$ at the input gates $v \in V_0$, to keep $C_1(v), C_0(v)$ alternate per layer. For the size of these sets use $c(v) = |C(v)|$.

Write $B_1(v) := \{X : f_v(X) = 0, \exists XY \in A(v)\}$ and $B_0(v) := \{Y : f_v(Y) = 0, \exists XY \in A(v)\}$ for the satisfying and unsatisfying assignments in a gate contained in at least one associated sensitive edge for that gate. For duality, write $B(v) = B_1(v)$ if v is a $\wedge$ gate and otherwise, when v is a $\vee$ gate use $B(v) = B_0(v)$. Write $b(v) = |B(v)|$ for the size of this set.

Unfortunately, lower-bounding the cones in one layer can only help lower-bound the satisfying/unsatisfying assignments on the following layer, making the induction fail. The nice assumption helps bridge this gap in the induction and assumes that each satisfying (or unsatisfying) assignment provides at least one 1- (or 0-) cone, so the elements of $B(v)$ are in bijection with the elements of $C(v)$.

A gate v is said to be nice if $b(v) = c(v)$. Note that in this case there is a natural bijection between the two. If $(X, D_{f_v}(X)) \in C_e(v)$, then $X \in B_e(v)$.

Example 62. *Consider the function f from the previous example.*



Suppose this function appears in a circuit, where the gate v computes exactly this function, $f = f_v$. f has 12 sensitive edges, suppose all of them are associated to v, giving $|A(v)| = 12$.

Suppose further that $\mathcal{L}(v) = \wedge$, meaning v is an AND gate in the circuit.

For $\wedge$ gates, $B(v)$ contains satisfying assignments next to an associated sensitive edge. Using the shortcut for small sets $\{a, b, c\} = abc$, the set $B(v)$ contains $\{0, 1, 01, 2, 3, 13, 013\}$. Notice that all satisfying assignment is next to a sensitive edge except $\emptyset$, so it is not included in $B(v)$.

$C(v)$ counts the 1-cones among the associated edges. In this case

$$C(v) = \{(0, 2), (1, 2), (01, 2), (2, 0), (3, 0), (013, 1)\}.$$

Unfortunately, this gate would not be nice. $13 \in B(v)$, but it is not a cone in $C(v)$, since $D_f(13) = 01$ not a singleton.

As a positive example, consider the same f and v scenario, but with $A(v)$ containing all sensitive edges of f except $(13, 123)$. In this case $B(v) = \{0, 1, 01, 2, 3, 013\}$ and $C(v)$ remains the same $\{(0, 2), (1, 2), (01, 2), (2, 0), (3, 0), (013, 1)\}$. Since they have the same size, this variant of the gate is nice.

Remark 63.

1. *Whether a gate is nice strongly depends on the subset of the sensitive edges associated to it and the underlying ordering of the inputs.*

2. *The block construction for PAR has every gate containing every possible sensitive edge, due to the lack of choices in the process of assigning the $A(v)$ associated sensitive edge sets. It is also easy to check that the block construction for PAR forms a nice circuit.*

8.4 Inductive Bound of Cones

The function $h(x) := x2^{1-x}$ will be important in the computations. Note that h is a bijection in the range $h : \{x : 2 \leq x\} \longrightarrow \{x : 0 < x \leq 1\}$. Write $h^{-1} : \{x : 0 < x \leq 1\} \longrightarrow \{x : 2 \leq x\}$ for the well-defined inverse in this region.

The main theorem of this chapter is the following.

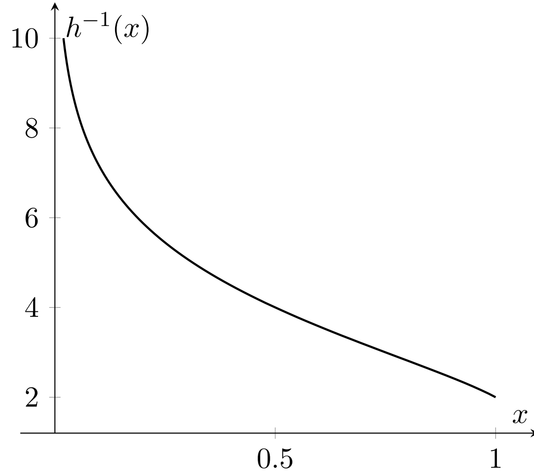


Figure 8.1: Plot of $h^{-1}(x)$.

Theorem 64. *Suppose a circuit $\mathcal{C}$ is given on n bits, depth d, sizes $m_i = |V_i|$ per layer and sensitivity $a(\mathcal{C}) = \alpha_0 n 2^{n-1}$. Suppose further that in $\mathcal{C}$ each gate is nice. Compute the values inductively $\alpha_{i+1} = \frac{\alpha_i}{h^{-1}(\alpha_i n / m_{i+1})}$, assuming that $m_{i+1} \geq \alpha_i n$ is satisfied for each $i < d$. Then it must hold that*

$$\alpha_d n 2^{n-1} \leq \sum_{v \in V_d} c(v).$$

The main inductive step is based on a modified version of the satisfiability coding lemma from [PPZ97].

Lemma 65. *Suppose a circuit on n inputs has $\alpha n \leq |V_{i+1}| = m$ for some α. Suppose, in addition, that*

$$\sum_{u \in V_i} c(u) \geq \alpha n 2^{n-1}.$$

Then

$$\sum_{v \in V_{i+1}} b(v) \geq \frac{\alpha}{h^{-1}(\alpha n / m)} n 2^{n-1}.$$

Unfortunately, this is not enough to provide an inductive application of the method. The condition that each gate is nice helps bridge this gap. Recall that the niceness condition at gate v is $b(v) = c(v)$.

Corollary 66. *Suppose a circuit on n inputs has $\alpha n \leq |V_{i+1}| = m$ for some α. Suppose, in addition, that*

$$\sum_{u \in V_i} c(u) \geq \alpha n 2^{n-1},$$

and each gate in V_{i+1} is nice. Then

$$\sum_{v \in V_{i+1}} c(v) \geq \frac{\alpha}{h^{-1}(\alpha n/m)} n 2^{n-1}.$$

Using this inductive statement, the main theorem follows.

Proof of Theorem 64. Count the number of cones in each layer $\sum_{v \in V_i} c(v)$. For the input layer, note that $\sum_{v \in V_0} c(v) = \sum_{v \in V_0} a(v) = a(\mathcal{C}) = \alpha_0 n 2^{n-1}$. The first equality comes from the fact that an input gate testing the bit $s \in [n]$ computes the function f_s, where any X has the smallest lexicographically closest change $D_{f_s}(X) = \{s\}$ a singleton. Hence every $(X, Y) \in A(s)$ has a corresponding cone. The second equality comes from the fact that the associated sensitive edge sets $\bigsqcup_{v \in V_i} A(v) = A(\mathcal{C})$ partition the total sensitive edges of the circuit in each layer. The theorem therefore follows from the inductive application of Corollary 66 for each layer. □

To get a size lower bound from this theorem, suppose that in a circuit a layer is smaller than expected ($m = |V_{i+1}|$ in the lemma statement), then the theorem shows the total cone count in the same layer and above will be larger than expected, resulting in a contradiction at the outputs, where the sensitive edges and the cones can be counted easily.

Remark 67. *The $\alpha n \leq m$ condition guarantees that h^{-1} is evaluated on $\alpha n/m \leq 1$, where it is well-defined. This function also captures the relations between consecutive layers of the block construction.*

With the notation of Theorem 21, consider the block circuit computing parity with parts $n = n_1 n_2 ... n_{d-1}$. Write $m_i = \frac{n}{n_1 n_2 ... n_i} 2^{n_i}$ and $\alpha_i = \frac{1}{n_1 n_2 ... n_i}$. Notice that these values satisfy a similar $\alpha_{i+1} = \frac{\alpha_i}{h^{-1}(2\alpha_i n/m_{i+1})}$ relation.

8.4.1 Application to Parity

This subsection contains the application of Theorem 64 to lower bound the size of nice circuits computing PAR. To compare with the lower-bound, recall the optimal construction for PAR.

Theorem 21. *Given an integer factoring of n as $n = n_1 n_2 ... n_{d-1}$, there exists a circuit computing* PAR *on n input bits having depth d and sizes $|V_i| = \frac{n}{n_1 n_2 ... n_i} 2^{n_i}$.*

Up to a constant factor, the same size lower-bound can be recovered for nice circuits.

Theorem 22. *Suppose a depth d, circuit $\mathcal{C}$ computes* PAR *on n bits, with each gate in $\mathcal{C}$ nice. Find a (not necessarily integer) sequence $n_1, n_2, ..., n_{d-1}$ such that $|V_i(\mathcal{C})| = \frac{n}{n_1 n_2 ... n_i} 2^{n_i - 1}$ for all $i \leq d$. If each $n_i \geq 2$, then it must hold that $\prod_i n_i \geq n$.*

Proof of Theorem 22. Without loss of generality assume the output gate is an $\vee$ gate. Remove the output gate of the circuit and note that the resulting depth $d-1$ circuit still computes a function with full sensitivity $a(\mathrm{PAR}) = n 2^{n-1}$, therefore $\alpha_0 = 1$. By the construction of the associated sensitive edges, $\bigsqcup_{v \in V_{d-1}} A(v) = G([n])$. For a fixed $v \in V_{d-1}$, any satisfying assignment X of f_v has $D_{f_o}(X) = \{0\}$ a possible cone. Every sensitive edge $X, X\Delta\{0\} \in A(v)$ is therefore an associated cone of v, meaning $(X, \{0\}) \in C(v)$ and correspondingly $\sum_{v \in V_{d-1}} c(v) = 2^{n-1}$, giving that $\alpha_{d-1} \leq \frac{1}{n 2^{n-1}} \sum_{v \in V_{d-1}} c(v) = 1/n$. Calculate the intermediate α_i values according to the statement of Theorem 64 and write $n_i = \alpha_{i-1}/\alpha_i$.

Using $m_i = |V_i|$, the construction of the α_i values imply that

$$\begin{array}{rclcll}
\alpha_{i+1} & = & \frac{\alpha_i}{h^{-1}\left(\frac{\alpha_i n}{m_{i+1}}\right)} & & & \Leftrightarrow \\
h^{-1}\left(\frac{\alpha_i n}{m_{i+1}}\right) & = & \frac{\alpha_i}{\alpha_{i+1}} & = & n_{i+1} & \Leftrightarrow \\
\frac{\alpha_i n}{m_{i+1}} & = & h(n_{i+1}) & = & n_{i+1} 2^{1-n_{i+1}} & \Leftrightarrow \\
m_{i+1} & = & \frac{\alpha_i n}{n_{i+1}} 2^{n_{i+1}-1} & = & \frac{n}{n_1 n_2 ... n_{i+1}} 2^{n_{i+1}-1} & .
\end{array}$$

Notice that $\alpha_i n / m_{i+1} = n_{i+1} 2^{1-n_{i+1}} \leq 1$ for each i, due to $n_i \geq 2$. The claim follows from $\prod n_i = 1/\alpha_{d-1} \geq n$, as required. □

Conditioning on fixed d, with n large enough, the optimal smallest total size $s = \sum_i |V_i|$

is attained when each layer $i < d$ has size approximately $m_i = s/(d-1) = \frac{n2^{n_i}}{n_1 n_2 \dots n_i}$. For a constant d, it asymptotically gives $m \approx 2^{n^{\frac{1}{d-1}}} n^{\frac{d-2}{2d-2}}$.

8.4.2 Proof of Lemma 65

Due to duality, one can assume, without loss of generality, that $\mathcal{L}(v) = \wedge$ for each $v \in V_{i+1}$. Notice that when $u \in V_i$, each $(Y, D) \in C_0(u)$ has an associated edge $XY = \{Y \Delta D, Y\} \in A(u)$, and the same edge must also appear in some $XY \in A(v)$ where $v \in V_{i+1}$, due to the definition of the $A(u), A(v)$ sets. As v is computing an $\wedge$ of other functions, including f_u, it must be the case that $(Y, D) \in C_0(v)$, therefore

$$\sum_{u \in V_i} |C_0(u)| \leq \sum_{v \in V_{i+1}} |C_0(v)|.$$

The following claim is based on the result in [PPZ97] and relates 0-cones in a function with the satisfying assignments. The proof of Claim 68 follows after this proof.

Claim 68. *For any $v \in V_{i+1}$, it is true that*

$$|C_0(v)| \leq |B_1(v)| \left(n - \log |B_1(v)|\right).$$

Using Claim 68, it holds that

$$\alpha n 2^{n-1} \leq \sum_{u \in V_i} |C_0(u)| \leq \sum_{v \in V_{i+1}} |C_0(v)| \leq \sum_{v \in V_{i+1}} |B_1(v)| \left(n - \log |B_1(v)|\right). \tag{8.2}$$

Writing $\tilde{b}(v) = \frac{b(v)}{n2^{n-1}}$ and using the shortcut $\phi = \sum_{v \in V_{i+1}} \tilde{b}(v)$, the above inequality becomes

$$\begin{aligned}
\alpha &\leq \sum_{v \in V_{i+1}} \tilde{b}(v)(1 - \log n - \log \tilde{b}(v)) = \\
&= \phi(1 - \log n) + \sum_{v \in V_{i+1}} -\tilde{b}(v) \log \tilde{b}(v) \leq \\
&\leq \phi(1 - \log n + \log m_{i+1} - \log \phi).
\end{aligned}$$

The first line follows from Equation (8.2) after normalizing by $n2^{n-1}$. The last line follows from the concavity of the entropy function $-x \log x$ and that there are $m_{i+1} = |V_{i+1}|$ many terms in the sum. Rearranging gives

$$\begin{aligned} \frac{\alpha}{\phi} &\leq 1 - \log n + \log m_{i+1} - \log \phi \\ \frac{n\phi}{m_{i+1}} &\leq 2^{1-\frac{\alpha}{\phi}} \\ \frac{\alpha n}{m_{i+1}} &\leq \frac{\alpha}{\phi} 2^{1-\alpha/\phi} = h(\alpha/\phi). \end{aligned}$$

The factors in each line (α, ϕ) are all positive, therefore the inequality is preserved. By assumption $\alpha n / m_{i+1} \leq 1$, therefore it lies in the region where h^{-1} is defined. As h is strictly decreasing in $\{x : x \geq 2\}$ it holds that $h^{-1}\left(\frac{\alpha n}{m_{i+1}}\right) \geq \frac{\alpha}{\phi}$ and correspondingly

$$\frac{\alpha}{h^{-1}\left(\frac{\alpha n}{m_{i+1}}\right)} n2^{n-1} \geq \phi n 2^{n-1} = \sum_{v \in V_{i+1}} c(v),$$

as claimed. The proof finishes with Claim 68, which is the version of [PPZ97] generalized to work on any function using cones.

Proof of Claim 68. Recall, the goal is to show

$$|C_0(v)| \leq |B_1(v)| \left(n - \log |B_1(v)|\right)$$

in a given function f_v.

- Partition all possible inputs $\{0,1\}^{[n]}$ into groups indexed by $B_1(v)$ using the rule

  $$P(X) := \left\{ Y : X = \operatorname{argmin}_{X' \in B_1(v)} Y \Delta X' \right\},$$

  giving

  $$\{0,1\}^{[n]} = \bigsqcup_{X \in B_1(v)} P(X).$$

- Partition the cones $C_0(v)$ into groups indexed by $B_1(v)$ using the rule

$$H(X) := \{(Y, D) \in C_0(v) : X = Y \Delta D\},$$

giving

$$C_0(v) = \bigsqcup_{X \in B_1(v)} H(X).$$

Notice that $H(X)$ is a proper partition, as every $(Y, D) \in C_0(v)$ must also have that $(Y, Y\Delta D) \in A(v)$, meaning that $Y\Delta D \in B_1(v)$. The fixed ordering is crucially used in this step. Both the cones and the partition $P(X)$ use the fixed ordering of the input bits in order to define the lexicographically smallest changes. The rest of the proof will argue that $|P(X)| \geq 2^{|H(X)|}$. Using this, the claim follows easily. The average partition size is $\frac{2^n}{|B_1(v)|}$ which can be bounded from below using

$$\frac{2^n}{|B_1(v)|} = \mathop{\mathbb{E}}_{X \in B_1(v)} |P(X)| \geq \mathop{\mathbb{E}}_{X \in B_1(v)} 2^{|H(X)|} \geq 2^{\mathbb{E}_{X \in B_1(v)} |H(X)|} = 2^{\frac{|C_0(v)|}{|B_1(v)|}},$$

where the second inequality used the convexity of 2^x. Taking logarithms yields the required inequality. In order to show $|P(X)| \geq 2^{|H(X)|}$, define a decision tree T for evaluating f_v as follows:

- Label the vertices by

$$V(T) = \left\{(X \cap [l], l) : X \in B_1(v), 0 \leq l \leq n\right\}.$$

- The edge $(S_1, l_1)(S_2, l_2)$ is present in $E(T)$ (where $(S_1, l_1), (S_2, l_2) \in V(T)$) if $l_1 = l_2 - 1$ and $S_1 = S_2 \cap [l_1]$.

Every $Y \in \{0, 1\}^{[n]}$ can be associated with a leaf by following the path from root, and at step l if there is a choice, then follow the direction that agrees based on $l \in Y$. Note that $Y \subseteq [n]$ stops at leaf (X, n) if $Y \in P(X)$, since every other choice would result in a lexicographically larger difference.

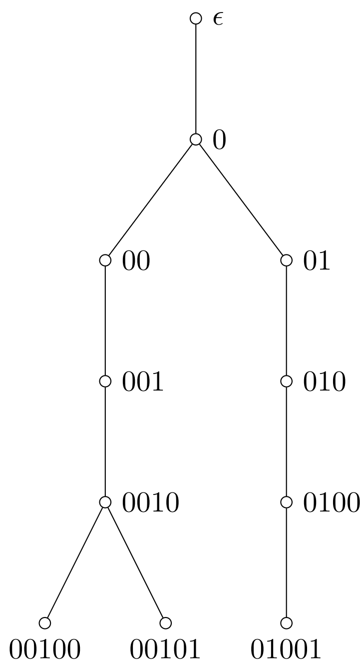


Figure 8.2: Example decision tree with boolean vectors instead of the sets. Instead of the label $(X \cap [l], l)$ use the l long initial segment of the boolean vector corresponding with X. The tree with $n = 5$ and $B_1(v) = \big\{\{2\}, \{2,4\}, \{1,4\}\big\} = \{00100, 00101, 01001\}$ is illustrated above.

Thus, when the vertex $(X \cap [l], l)$ in the decision tree has only one descendant, say $(X \cap [l+1], l+1)$, then any $Y \in \{0,1\}^{[n]}$ input with matching initial segment $Y \cap [l] = X \cap [l]$ must associate to an element from $B_1(v)$ with the same $X \cap [l+1]$ continuation. For $X \in B_1(v)$, write $H'(X)$ to be the number of l, such that there is no branching at $(X \cap [l], l)$.

Notice that $P(X)$ is exactly the set of inputs that agree with X, outside the $H'(X)$ bits, where they can be arbitrary. In particular, $|P(X)| = 2^{|H'(X)|}$. Furthermore, each cone $(X\Delta\{l\}, \{l\}) \in H(X)$ means that there are no satisfying assignments with initial segment $(X\Delta\{l, l+1\}, l+1)$, hence there is no branching at the vertex $(X \cap [l], l)$. Therefore $|H(X)| \leq |H'(X)|$ and the claim follows. $\square$

Remark 69. *The proof of Claim 68 used the ordering of the inputs. The decision process that placed each input Y into the corresponding partition $P(X)$ was based on the testing the elements of Y in the fixed order. Each cone in $H(X)$ provided a lack of choice in this process,*

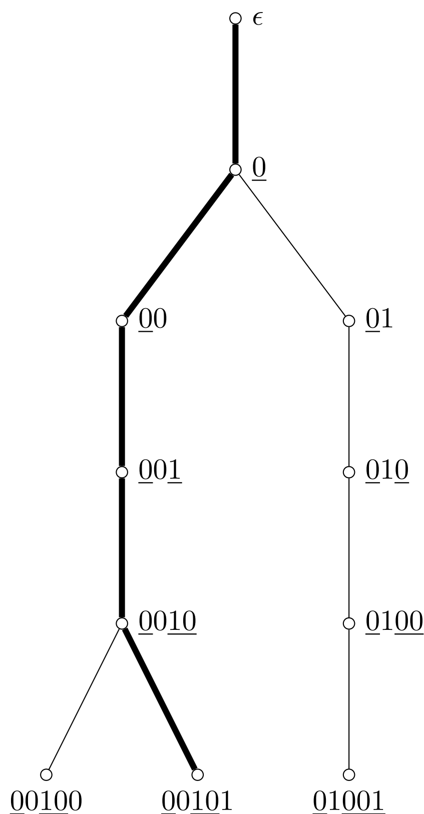


Figure 8.3: Continuing the same example with the same boolean vector notation, underline the input bits where there is no branching. The path of $Y = \{0, 2, 3, 4\} = 10111$ is highlighted, ending at the leaf 00101.

similar to critical clauses helping encode the satisfying assignments in [PPZ97].

8.5 Concluding Remarks

This chapter primarily aimed to establish a connection between inductive arguments relying on circuit depth and the satisfiability coding lemma. A combinatorial characterization of the coding lemma in terms of associated cones and associated sensitive edges was provided. Furthermore, under a niceness assumption, it was demonstrated that a coding-lemma-style argument can yield tight constant-depth circuit-size lower bounds for the PAR problem. However, whether the size lower bound holds unconditionally remains an open question.

Conjecture 70. *Every depth d circuit computing* $\{\text{PAR}, \neg\,\text{PAR}\}$ *must have a product* $n \leq$

$n_1 n_2 ... n_{d-1}$, *such that* $|V_i| = \frac{n}{n_1 n_2 ... n_i} 2^{n_i}$.

The argument assumes an ordering of the inputs, but only when defining the cones based on the lexicographically smallest difference. This assumption makes it work with any ordering, provided the niceness conditions are still met.

Conjecture 71. *Suppose a circuit* $\mathcal{C}$ *is given on* n *input bits with depth* d*, sizes* $m_i = |V_i|$ *per layer, and total sensitivity* $a(\mathcal{C}) = \alpha_0 n 2^{n-1}$*. Write* $\alpha_{i+1} = \frac{\alpha_i}{h^{-1}(\alpha_i n / m_{i+1})}$ *(assuming* $m_{i+1} \geq \alpha_i n$*) for each* $i < d$*. Then it holds that*

$$\alpha_2 n 2^{n-1} \leq \sum_{v \in V_2} c(v).$$

This conjecture would help establish a tight unconditional depth 4 size lower bound for circuits computing PAR.

Bibliography

[AB87] N. Alon and R. B. Boppana. The monotone circuit complexity of Boolean functions. *Combinatorica*, 7(1):1–22, Mar 1987.

[AB09] S. Arora and B. Barak. *Computational Complexity: A Modern Approach*, chapter 23. Cambridge University Press, 2009.

[Ajt83] Miklós Ajtai. Σ_1^1-Formulae on Finite Structures. *Annals of Pure and Applied Logic*, 24(1):1, 1983.

[AM] K. Amano and A. Maruoka. A superpolynomial lower bound for a circuit computing the clique function with at most (1/6) log log n negation gates. In *Mathematical Foundations of Computer Science 1998*, pages 399–408. Springer Berlin Heidelberg.

[Ama11] Kazuyuki Amano. Tight Bounds on the Average Sensitivity of k-CNF. *Theory of Computing*, 7(4):45–48, 2011.

[Ama23] Kazuyuki Amano. Depth-Three Circuits for Inner Product and Majority Functions. In *34th International Symposium on Algorithms and Computation (ISAAC 2023)*, volume 283 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 7:1–7:16, 2023.

[AS16] N. Alon and C. Shikhelman. Many T copies in H-free graphs. *Journal of Combinatorial Theory, Series B*, 121:146–172, 2016. Fifty years of The Journal of Combinatorial Theory.

[AW08] S. Aaronson and A. Wigderson. Algebrization: a new barrier in complexity theory. In *Proceedings of the Fortieth Annual ACM Symposium on Theory of Computing*, STOC '08, page 731–740. Association for Computing Machinery, 2008.

[Bab11] Rahil Baber. *Some Results in Extremal Combinatorics*. PhD thesis, University College London, 2011.

[Bea90] Paul Beame. Lower Bounds for Recognizing Small Cliques on CRCW PRAM'S. *Discrete Appl. Math.*, 29(1):3–20, November 1990.

[BGS75] T. Baker, J. Gill, and R. Solovay. Relativizations of the P =? NP Question. *SIAM Journal on Computing*, 4(4):431–442, 1975.

[BHLP16] J. Balogh, P. Hu, B. Lidický, and F. Pfender. Maximum density of induced 5-cycle is achieved by an iterated blow-up of 5-cycle. *European Journal of Combinatorics*, 52:47–58, 2016.

[BHST14] E. Blais, J. Håstad, R. A. Servedio, and L. Tan. On DNF Approximators for Monotone Boolean Functions. In *Automata, Languages, and Programming*, pages 235–246, Berlin, Heidelberg, 2014.

[Bop86] Ravi B. Boppana. Threshold functions and bounded depth monotone circuits. *Journal of Computer and System Sciences*, 32(2):222–229, 1986.

[Bor99] Brian Borchers. CSDP, A C library for semidefinite programming. *Optimization Methods and Software*, 11(1-4):613–623, 1999.

[BP14] J. Balogh and Š. Petříčková. The number of the maximal triangle-free graphs. *Bulletin of the London Mathematical Society*, 46(5):1003–1006, July 2014.

[BRC60] R.C. Bose and D.K. Ray-Chaudhuri. On a class of error correcting binary group codes. *Information and Control*, 3(1):68–79, 1960.

[CH82] Ashok Chandra and David Harel. Structure and complexity of relational queries. *Journal of Computer and System Sciences*, 25(1):99–128, 1982.

[CKR20] B. P. Cavalar, M. Kumar, and B. Rossman. Monotone Circuit Lower Bounds from Robust Sunflowers, 2020.

[CL99] F. Chung and L. Lu. An upper bound for the Turán number $t_3(n, 4)$. *J. Comb. Theory Ser. A*, 87(2):381–389, aug 1999.

[CL24] W. Chen and X. Liu. Strong stability from vertex-extendability and applications in generalized Turán problems, 2024.

[CO23] Bruno P. Cavalar and Igor C. Oliveira. Constant-Depth Circuits vs. Monotone Circuits. In *38th Computational Complexity Conference (CCC 2023)*, volume 264 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 29:1–29:37, 2023.

[Dan96] Vlado Dančík. Complexity of boolean functions over bases with unbounded fan-in gates. *Information Processing Letters*, 57(1):31–34, 1996.

[dC83] D. de Caen. Extension of a theorem of Moon and Moser on complete subgraphs, 1983.

[ER60] P. Erdös and R. Rado. Intersection theorems for systems of sets. *Journal of the London Mathematical Society*, s1-35(1):85–90, 1960.

[Erd62] Paul Erdős. On the number of complete subgraphs contained in certain graphs. *Magyar Tud. Akad. Mat. Kut. Int. Kozl*, 7:459–474, 1962.

[Erd84] Paul Erdős. On some problems in graph theory, combinatorial analysis and combinatorial number theory. *Graph theory and combinatorics (Cambridge, 1983)*, pages 1–17, 1984.

[ES46] P. Erdős and A. H. Stone. On the structure of linear graphs. *Bulletin of the American Mathematical Society*, 52(12):1087 – 1091, 1946.

[Fag74] Ronald Fagin. Generalized first-order spectra, and polynomial time recognizable sets. *SIAM-AMS Proc.*, 7, 01 1974.

[FGT22] P. Frankl, S. Gryaznov, and N. Talebanfard. A Variant of the VC-Dimension with Applications to Depth-3 Circuits. In *ITCS 2022*, volume 215 of *LIPIcs*, pages 72:1–72:19, 2022.

[FRV11] V. Falgas-Ravry and E. R. Vaughan. On applications of Razborov's flag algebra calculus to extremal 3-graph theory. *arXiv*, 2011.

[FSS81] M. Furst, J. B. Saxe, and M. Sipser. Parity, Circuits, and the Polynomial-Time Hierarchy. In *Proceedings of the 22nd Annual Symposium on Foundations of Computer Science*, SFCS '81, page 260–270, 1981.

[FSV93] R. Fagin, L. Stockmeyer, and M.Y. Vardi. On monadic np vs. monadic co-np. In *[1993] Proceedings of the Eigth Annual Structure in Complexity Theory Conference*, pages 19–30, 1993.

[GGKS20] Ankit Garg, Mika Göös, Pritish Kamath, and Dmitry Sokolov. Monotone circuit lower bounds from resolution. *Theory of Computing*, 16(13):1–30, 2020.

[GGM23] Mika Göös, Ziyi Guan, and Tiberiu Mosnoi. Depth-3 Circuits for Inner Product. In *48th International Symposium on Mathematical Foundations of Computer Science (MFCS 2023)*, volume 272, pages 51:1–51:12, 2023.

[GP18] Mika Göös and Toniann Pitassi. Communication lower bounds via critical block sensitivity. *SIAM Journal on Computing*, 47(5):1778–1806, jan 2018.

[GPP+24] M. Gurumukhani, M. Paturi, P. Pudlák, M. Saks, and N. Talebanfard. Local Enumeration and Majority Lower Bounds, 2024.

[GRSS24] Mika Göös, Artur Riazanov, Anastasia Sofronova, and Dmitry Sokolov. Top-Down Lower Bounds for Depth-Four Circuits, 2024.

[HJP95] J. Håstad, S. Jukna, and P. Pudlak. Top-down lower bounds for depth-three circuits. *Computational Complexity*, 5(2):99–112, June 1995.

[HM10] S. Heubach and T. Mansour. *Combinatorics of compositions and words [electronic resource].* Discrete mathematics and its applications. CRC Press, Boca Raton, 2010.

[Hå86] Johan Håstad. Almost optimal lower bounds for small depth circuits. In *Proceedings of the eighteenth annual ACM symposium on Theory of computing - STOC '86*, pages 6–20, Berkeley, California, United States, 1986. ACM Press.

[Hå14] Johan Håstad. On the Correlation of Parity and Small-Depth Circuits. *SIAM Journal on Computing*, 43:1699–1708, 01 2014.

[Hå24] Johan Håstad. On Small-depth Frege Proofs for PHP, 2024.

[Imm82] Neil Immerman. Relational queries computable in polynomial time (extended abstract). In *Proceedings of the Fourteenth Annual ACM Symposium on Theory of Computing*, STOC '82, page 147–152. Association for Computing Machinery, 1982.

[Imm87] Neil Immerman. Languages that capture complexity classes. *SIAM Journal on Computing*, 16(4):760–778, 1987.

[Imm88] Neil Immerman. Nondeterministic space is closed under complementation. *SIAM Journal on Computing*, 17(5):935–938, 1988.

[Imm95] Neil Immerman. Descriptive complexity: A logician's approach to computation. *Notices of the American Mathematical Society*, 42(10):1127–1133, 1995.

[Imm98] Neil Immerman. *Descriptive complexity.* Springer Science & Business Media, 1998.

[IPZ01] Russell Impagliazzo, Ramamohan Paturi, and Francis Zane. Which problems have strongly exponential complexity? *Journal of Computer and System Sciences*, 63(4):512–530, dec 2001.

[JK07] T. Junttila and P. Kaski. Engineering an efficient canonical labeling tool for large and sparse graphs. In *Proceedings of the Ninth Workshop on Algorithm Engi-*

neering and Experiments and the Fourth Workshop on Analytic Algorithms and Combinatorics, pages 135–149. SIAM, 2007.

[Kee11] Peter Keevash. *Hypergraph Turán problems*, page 83–140. London Mathematical Society Lecture Note Series. Cambridge University Press, 2011.

[KW90] Mauricio Karchmer and Avi Wigderson. Monotone circuits for connectivity require super-logarithmic depth. *SIAM Journal on Discrete Mathematics*, 3(2):255–265, may 1990.

[LMN93] Nathan Linial, Yishay Mansour, and Noam Nisan. Constant depth circuits, fourier transform, and learnability. *J. ACM*, 40(3):607–620, July 1993.

[LP21] B. Lidický and F. Pfender. Semidefinite programming and Ramsey numbers. *SIAM Journal on Discrete Mathematics*, 35(4):2328–2344, 2021.

[LRT22] V. Lecomte, P. Ramakrishnan, and L. Tan. The composition complexity of majority, 2022.

[Lyn86] James F. Lynch. A depth-size tradeoff for boolean circuits with unbounded fan-in. In *Structure in Complexity Theory*, pages 234–248, Berlin, Heidelberg, 1986. Springer Berlin Heidelberg.

[Mal] Ash Malyshev. Diagonal asymptotics of integer compositions. MathOverflow. URL:https://mathoverflow.net/q/191567 (version: 2014-12-27).

[Man07] Willem Mantel. Problem 28. *Wiskundige Opgaven*, 10:60–61, 1907.

[MP14] B. D. McKay and A. Piperno. Practical graph isomorphism, ii. *Journal of Symbolic Computation*, 60:94–112, 2014.

[MST22] M. Mossé, H. Sha, and L. Tan. A Generalization of the Satisfiability Coding Lemma and Its Applications. In *25th International Conference on Theory and Applications of Satisfiability Testing (SAT 2022)*, volume 236, pages 9:1–9:18, 2022.

[Mub06] Dhruv Mubayi. A hypergraph extension of Turán's theorem. *Journal of Combinatorial Theory, Series B*, 96(1):122–134, 2006.

[MW19] Or Meir and Avi Wigderson. Prediction from partial information and hindsight, with application to circuit lower bounds. *Computational Complexity*, 28(2):145–183, 2019.

[O'D07] O'Donnell, R. and Wimmer, K. Approximation by DNF: Examples and Counterexamples. In *Automata, Languages and Programming*, pages 195–206, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[PBI93] Toniann Pitassi, Paul Beame, and Russell Impagliazzo. Exponential lower bounds for the pigeonhole principle. *computational complexity*, 3(2):97–140, Jun 1993.

[PPSZ05] R. Paturi, P. Pudlák, M. E. Saks, and F. Zane. An improved exponential-time algorithm for k-SAT. *J. ACM*, 52(3):337–364, may 2005.

[PPZ97] R. Paturi, P. Pudlak, and F. Zane. Satisfiability Coding Lemma. In *Proceedings of the 38th Annual Symposium on Foundations of Computer Science*, FOCS '97, page 566, USA, 1997. IEEE Computer Society.

[PR17] T. Pitassi and R. Robere. Strongly exponential lower bounds for monotone computation. In *Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing - STOC 2017*, pages 1246–1255, Montreal, Canada, 2017. ACM Press.

[PRST16] Toniann Pitassi, Benjamin Rossman, Rocco A. Servedio, and Li-Yang Tan. Polylogarithmic frege depth lower bounds via an expander switching lemma. In *Proceedings of the Forty-Eighth Annual ACM Symposium on Theory of Computing*, STOC '16, page 644–657. Association for Computing Machinery, 2016.

[PRT22] Toniann Pitassi, Prasanna Ramakrishnan, and Li-Yang Tan. Tradeoffs for small-depth Frege proofs, 2022.

[PSZ00] Ramamohan Paturi, Michael Saks, and Francis Zane. Exponential lower bounds for depth three boolean circuits. *computational complexity*, 9(1):1–15, 2000.

[Pá41] Pál Turán. Eine extremalaufgabe aus der graphentheorie. *Mat. és Fiz. Lapok*, 48:436–452, 1941.

[Rat08] Joel Ratsaby. Estimate of the number of restricted integer-partitions. *Applicable Analysis and Discrete Mathematics*, 2(2):222–233, 2008.

[Raz85] Alexander A. Razborov. Lower Bounds on the Monotone Complexity of Some Boolean Function. *Soviet Math. Dokl.*, 31:354–357, 1985.

[Raz07] Alexander A. Razborov. Flag Algebras. *The Journal of Symbolic Logic*, 72(4):1239–1282, 2007.

[Raz10] Alexander A. Razborov. On 3-Hypergraphs with Forbidden 4-Vertex Configurations. *SIAM J. Discret. Math.*, 24(3):946–963, August 2010.

[Raz13] Alexander A. Razborov. Flag Algebras: An Interim Report. In *The Mathematics of Paul Erdős II*, 2013.

[Ros08] Benjamin Rossman. On the constant-depth complexity of k-clique. In *Proceedings of the fourtieth annual ACM symposium on Theory of computing - STOC 08*, page 721, Victoria, British Columbia, Canada, 2008. ACM Press.

[Ros17] Benjamin Rossman. An entropy proof of the switching lemma and tight bounds on the decision-tree size of ac 0. 2017.

[Ros18] Benjamin Rossman. The Average Sensitivity of Bounded-Depth Formulas. *Comput. Complex.*, 27(2):209–223, June 2018.

[RR97] A. A Razborov and S. Rudich. Natural Proofs. *J. Comput. Syst. Sci.*, 55(1):24–35, August 1997.

[RW92] Ran Raz and Avi Wigderson. Monotone circuits for matching require linear depth. *Journal of the ACM*, 39(3):736–744, 1992.

[Sau71] N. Sauer. A generalization of a theorem of Turán. *Journal of Combinatorial Theory, Series B*, 10(2):109–112, 1971.

[Ser19] Igor S. Sergeev. On the complexity of bounded-depth circuits and formulas over the basis of unbounded fan-in gates. *Discrete Mathematics and Applications*, 29:241–254, 08 2019.

[SFS16] M. K. Silva, F. M. Filho, and C. M. Sato. Flag Algebras: A First Glance, 2016.

[Sid87] Alexander F. Sidorenko. Precise values of Turán numbers. *Mathematical Notes of the Academy of Sciences of the USSR*, 42(5):913–918, November 1987.

[Sid95] Alexander F. Sidorenko. What we know and what we do not know about Turán numbers. *Graphs and Combinatorics*, 11:179–199, 06 1995.

[SS18] J. Sliacan and W. Stromquist. Improving bounds on packing densities of 4-point permutations. *Discrete Mathematics & Theoretical Computer Science*, Vol. 19 no. 2, Permutation Patterns 2016, February 2018.

[ST18] Alexander V. Smal and Navid Talebanfard. Prediction from partial information and hindsight, an alternative proof. *Information Processing Letters*, 136:102–104, 2018.

[Sze88] Róbert Szelepcsényi. The method of forced enumeration for nondeterministic automata. *Acta Informatica*, 26:279–284, 1988.

[Tal17] Avishay Tal. Tight Bounds on the Fourier Spectrum of AC0. In Ryan O'Donnell, editor, *32nd Computational Complexity Conference (CCC 2017)*, volume 79 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 15:1–15:31, 2017.

[The24] The Sage Developers. *SageMath, the Sage Mathematics Software System (Version 10.3)*, 2024. `https://www.sagemath.org`.

[Usm94] Riaz A. Usmani. Inversion of a tridiagonal Jacobi matrix. *Linear Algebra and its Applications*, 212-213:413–414, 1994.

[Val] Leslie G. Valiant. Graph-theoretic arguments in low-level complexity. In Jozef Gruska, editor, *Mathematical Foundations of Computer Science 1977*, pages 162–176, Berlin, Heidelberg. Springer Berlin Heidelberg.

[Var82] Moshe Y. Vardi. The complexity of relational query languages (extended abstract). In Harry R. Lewis, Barbara B. Simons, Walter A. Burkhard, and Lawrence H. Landweber, editors, *Proceedings of the 14th Annual ACM Symposium on Theory of Computing, 1982*, pages 137–146. ACM, 1982.

[XZG21] Z. Xu, T. Zhang, and G. Ge. Some extremal results on hypergraph Turán problems. *Science China Mathematics*, 65:1765–1774, 2021.

[Yao85] Andrew C-C. Yao. Separating the polynomial-time hierarchy by oracles. In *Proc. 26th Annual Symposium on Foundations of Computer Science*, page 1–10. IEEE Press, 1985.

[Zyk49] Alexander Aleksandrovich Zykov. On some properties of linear complexes. *Matematicheskii sbornik*, 66(2):163–188, 1949.

Appendix A

Flag Algebra Code

This section contains the code written to perform flag algebraic calculations in SAGE [The24], the open source mathematical software.

A.1 Combinatorial Theory, Flag Algebra, Flag Algebra Element

```
1 r"""
2 Implementation of flag algebras, with a class for combinatorial theories
3
4
5 A combinatorial theory is any theory with universal axioms only,
6 (therefore the elements satisfy a heredetary property). This
7 implementation allows the construction of any such theory, and
8 can perform flag algebraic computations on them. The theory of
9 flag algebras is from [Raz2007]_
10
11 To find out more about flags, how to create and manipulate them,
12 see :mod:`sage.algebras.flag`. This docstring is for combinatorial
13 theories and combinatorial optimization problems using flag algebras.
14
15 The examples will use ::
```

```
16
17     sage: from sage.algebras.flag_algebras import *
18
19 To create a `CombinatorialTheory` object we need to first know the signature.
20 For example graphs have one relational symbol for the edges, of arity 2.
21 We could provide for example `edges=2` as a parameter to the constructor,
22 showing this fact.
23
24 In addition we need two important functions:
25
26 A generator function, showing the program how to generate elements of
27 the theory for a fixed size. For example, to graphs with ordered vertices (
28 or equivalently 0-1 symmetric matrices with 0 diagonal) can be generated
29 with the following code ::
30
31     sage: def test_generator_ov_graph(n):
32     ....:     full = list(itertools.combinations(range(n), int(2)))
33     ....:     for ii in range(binomial(n, 2)+1):
34     ....:         for xx in itertools.combinations(full, int(ii)):
35     ....:             yield {'edges': xx}
36
37
38 This function takes `n` integer as input and returns the possible
39 ways one can construct such graphs with ordered vertices. Note
40 the elements returned are dictionaries. The key is `'edges'` and
41 the value is the actual list of edges. `'edges'` key is important to
42 match the one defined in the signature.
43
44 The second important function is the identifier function. This takes
45 as input an element of the theory, with some of the points marked,
46 and returns something that uniquely identifies this element, up to
47 automorphism. The automorphisms must respect the marked points. For
48 ordered vertex graphs, since there is no automorphism of the vertices
49 giving the exact same order this is easy, we can just return the edges
50 sorted with the marked points and the total number of points. ::
```

```
51
52      sage: def test_identify_ov_graph(n, ftype_points, edges):
53      ....:     return (n, tuple(ftype_points), \
54      ....:     tuple(sorted(list(edges))))
55      ....:
56
57  For theories where members can have more automorphisms, this might be
58  harder. Then to create the theory, provide a name, the two functions and
59  the signature ::
60
61      sage: TestOVGraphTheory = CombinatorialTheory('TestOVGraph', \
62      ....: test_generator_ov_graph, test_identify_ov_graph, edges=2)
63
64  The following common theories are already implemented:
65  -GraphTheory
66  -ThreeGraphTheory
67  -DiGraphTheory
68  -TournamentTheory
69  -PermutationTheory
70  -OVGraphTheory (graphs with ordered vertices)
71  -OEGraphTheory (graphs with ordered edges)
72  -RamseyGraphTheory (see [LiPf2021]_ for explanation)
73
74  The rest of this docstring will use `GraphTheory` since the number
75  of structures is realtively small there. `Flag` docstring shows
76  ways to create and calculate with flags. To create a flag we need to
77  provide the vertex size and a list of elements for each signature.
78  To create an edge flag for graphs, use ::
79
80      sage: e = GraphTheory(2, edges=[[0, 1]])
81
82  To create a triangle `K_3` we can write ::
83
84      sage: k3 = GraphTheory(3, edges=[[0, 1], [0, 2], [1, 2]])
85
```

```
86 If we have a theory, say GraphTheory, we can simply exclude structures
87 with :func:`exclude`. This takes a list of flags or a single flag ::
88
89     sage: GraphTheory.exclude(k3)
90
91 Excluding structures overwrites the theory, and in the future will only
92 consider members without the excluded structures. So with the above,
93 excluding a triangle will make GraphTheory not generate `k3`, or any larger
94 flag with induced `k3` in it. We can check this by generating flags of size
95 `4` ::
96
97     sage: GraphTheory.generate_flags(4)
98     (Flag on 4 points, ftype from [] with edges=[],
99      Flag on 4 points, ftype from [] with edges=[[0, 3]],
100      Flag on 4 points, ftype from [] with edges=[[0, 3], [1, 3]],
101      Flag on 4 points, ftype from [] with edges=[[0, 3], [1, 3], [2, 3]],
102      Flag on 4 points, ftype from [] with edges=[[0, 2], [1, 3]],
103      Flag on 4 points, ftype from [] with edges=[[0, 2], [0, 3], [1, 3]],
104      Flag on 4 points, ftype from [] with edges=[[0, 2], [0, 3], [1, 2], [1,
    3]])
105
106 Excluding structures overwrites the previously excluded structures. Calling
107 `exclude` without any arguments excludes nothing, giving back the original
108 theory.
109
110 To optimize the density of a flag (or linear combination of flags) in a theory
    ,
111 we can call :func:`optimize_problem`. To try to find the maximum number of
112 edges `e` in `k3` free graphs we can write ::
113
114     sage: x = GraphTheory.optimize_problem(e, 3)
115     ...
116     Optimal solution found.
117     sage: abs(x-0.5)<1e-6
118     True
```

```
119
120 The second parameter, `optimize_problem(e, 3)` indicates the maximum size the
121 program expands the flags. We can reset the excluded graphs, and try to
      minimize
122 the density of triangles and empty triples ::
123
124     sage: GraphTheory.exclude()
125     sage: e3 = GraphTheory(3)
126     sage: x = GraphTheory.optimize_problem(e3+k3, 3, maximize=False)
127     ...
128     sage: abs(x-0.25)<1e-6
129     True
130
131 The `solve_problem` function requires csdpy, an sdp solver. But it is possible
132 to get these relations directly, by expressing the inequalities
133 as a sum of squares ::
134
135     sage: GraphTheory.exclude(k3)
136     sage: pe = GraphTheory(2, ftype=[0]) - 1/2
137     sage: 1/2 >= pe.mul_project(pe) * 2 + e
138     True
139
140 :func:`mul_project` is short for multiplication and projection, the
141 non-negativity of self multiplication is preserved after projection so
142 `pe.mul_project(pe)` is non-negative on all large enough structures
143 giving that `1/2` is larger than `e` in all large enough structures.
144
145 The following longer example shows that the density of K^3_4 is always
146 less than 3/8 in K^3_5-free hypergraphs. It uses the ThreeGraphTheory
147 object to deal with 3-uniform hypergraphs and hand-picked squares.
148 These values come from [Bod2023]_::
149
150     sage: em5 = ThreeGraphTheory(5, edges=[])
151     sage: ThreeGraphTheory.exclude(em5)
152
```

```
153    sage: em4 = ThreeGraphTheory(4, edges=[])
154
155    sage: em3p1 = ThreeGraphTheory(3, ftype_points=[0], edges=[])
156    sage: sq1 = (em3p1 - 3/4).mul_project(em3p1 - 3/4) # todo: not implemented
157
158    sage: la4p2 = ThreeGraphTheory(4, ftype_points=[0, 1], edges=[[0, 2, 3]])
159    sage: lb4p2 = ThreeGraphTheory(4, ftype_points=[0, 1], edges=[[1, 2, 3]])
160    sage: sq2 = (la4p2 - lb4p2).mul_project(la4p2 - lb4p2) # todo: not
       implemented
161
162    sage: ma4p3 = ThreeGraphTheory(4, ftype_points=[0, 1, 2], edges=[[0, 1,
       3]])
163    sage: mb4p3 = ThreeGraphTheory(4, ftype_points=[0, 1, 2], edges=[[0, 2,
       3]])
164    sage: mc4p3 = ThreeGraphTheory(4, ftype_points=[0, 1, 2], edges=[[1, 2,
       3]])
165    sage: sq3 = (ma4p3 + mb4p3 + mc4p3 - 1/2).mul_project(ma4p3 + mb4p3 +
       mc4p3 - 1/2) # todo: not implemented
166
167    sage: em4p3 = ThreeGraphTheory(4, ftype_points=[0, 1, 2])
168    sage: sq4 = (em4p3 - 1/2).mul_project(em4p3 - 1/2) # todo: not implemented
169
170    sage: n5q4 = ThreeGraphTheory(5, ftype_points=[0, 1, 2, 3], edges=[[0, 1,
       2]])
171    sage: sq5 = (n5q4 - 1/2).mul_project(n5q4 - 1/2) # todo: not implemented
172
173    sage: oa5t4 = ThreeGraphTheory(5, ftype_points=[0, 1, 2, 3], edges=[[0, 1,
        4]])
174    sage: ob5t4 = ThreeGraphTheory(5, ftype_points=[0, 1, 2, 3], edges=[[2, 3,
        4]])
175    sage: sq6 = (oa5t4 - ob5t4).mul_project(oa5t4 - ob5t4) # todo: not
       implemented
176
177    sage: sos = 2/3 * sq1 + 1/6 * sq2 + 13/12 * sq3 + 11/12 * sq4 + 2 * sq5 +
       1/2 * sq6 # todo: not implemented
```

```
178     sage: 3/8 >= sos + em4 # todo: not implemented
179     True
180
181
182 .. SEEALSO::
183     :func:`CombinatorialTheory.__init__`
184     :func:`CombinatorialTheory.exclude`
185     :func:`CombinatorialTheory.optimize_problem`
186     :func:`CombinatorialTheory.generate_flags`
187
188 AUTHORS:
189
190 - Levente Bodnar (Dec 2023): Initial version
191
192 """
193
194
195 from functools import lru_cache
196 import itertools
197
198 from sage.structure.richcmp import richcmp
199 from sage.structure.unique_representation import UniqueRepresentation
200 from sage.structure.parent import Parent
201 from sage.structure.element import CommutativeAlgebraElement
202 from sage.rings.ring import CommutativeAlgebra
203 from sage.rings.rational_field import QQ
204 from sage.rings.real_mpfr import RR
205 from sage.algebras.flag import Flag
206
207 from sage.categories.sets_cat import Sets
208
209 from sage.modules.free_module_element import vector
210 from sage.matrix.constructor import matrix
211
212 from sage.misc.prandom import randint
```

```
213 from sage.arith.misc import falling_factorial, binomial
214
215 from sage.graphs.graph import Graph
216 from sage.graphs.digraph import DiGraph
217 from sage.misc.lazy_import import lazy_import
218 lazy_import("sage.graphs.graph_generators", "graphs")
219 lazy_import("sage.graphs.digraph_generators", "digraphs")
220 lazy_import("sage.graphs.hypergraph_generators", "hypergraphs")
221
222 import pickle
223 import os
224
225 class CombinatorialTheory(Parent, UniqueRepresentation):
226
227     Element = Flag
228
229     def __init__(self, name, generator, identifier, **signature):
230         r"""
231         Initialize a Combinatorial Theory
232
233         A combinatorial theory is any theory with universal axioms only,
234         (therefore the elements satisfy a heredetary property).
235         See the file docstring for more information.
236
237         INPUT:
238
239         - ``name`` -- string; name of the Theory
240         - ``generator`` -- function; generates elements
241             of the theory. For a given input ``n``
242             returns a list of elements of the theory
243             in a dictionary format (for each
244             value in the signature, one dictionary
245             entry describing the blocks corresponding to
246             that signature)
247         - ``identifier`` -- function; given a structure
```

```
248             with the matching signature of this theory,
249             returns a unique identifier, such that
250             automorphic structures return the same
251             value.
252         - ``**signature`` -- named integers; the signature
253             of the theory, for each name a corresponding number
254             giving the arity of that symbol
255 
256         OUTPUT: A CombinatorialTheory object
257 
258         EXAMPLES::
259 
260         This example shows how to create the theory for graphs
261         with ordered vertices (or equivalently 0-1 matrices)::
262 
263             sage: from sage.algebras.flag_algebras import *
264             sage: def test_generator_ov_graph(n):
265             ....:     full = list(itertools.combinations(range(n), int(2)))
266             ....:     for ii in range(binomial(n, 2)+1):
267             ....:         for xx in itertools.combinations(full, int(ii)):
268             ....:             yield {'edges': xx}
269             ....:
270             sage: def test_identify_ov_graph(n, ftype_points, edges):
271             ....:     return (n, tuple(ftype_points), \
272             ....:     tuple(sorted(list(edges))))
273             ....:
274             sage: TestOVGraphTheory = CombinatorialTheory('TestOVGraph', \
275             ....: test_generator_ov_graph, test_identify_ov_graph, edges=2)
276             sage: TestOVGraphTheory
277             Theory for TestOVGraph
278 
279         .. NOTE::
280 
281             There are pre-constructed CombinatorialTheory objects
282             in sage.algebras.flag_algebras for the following:
```

```
283             -GraphTheory
284             -ThreeGraphTheory
285             -DiGraphTheory
286             -TournamentTheory
287             -PermutationTheory
288             -OVGraphTheory (graphs with ordered vertices)
289             -OEGraphTheory (graphs with ordered edges)
290             -RamseyGraphTheory (see [LiPf2021]_ for explanation)
291         """
292         self._signature = signature
293         self._excluded = []
294         self._generator = generator
295         self._identifier = identifier
296         self._name = name
297         Parent.__init__(self, category=(Sets(), ))
298         self._populate_coercion_lists_()
299 
300     def _compress(self, numbers):
301         r"""
302         Compresses the list of numbers to a short string.
303 
304         This is used when naming files in saving and loading, to store the
    parameters of this theory.
305 
306         EXAMPLES::
307 
308             sage: from sage.algebras.flag_algebras import *
309             sage: GraphTheory._compress([1, 1, 10, 4, 2, 11, 15])
310             '_kKgb'
311         """
312         table = "
    abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789-_"
313 
314         p = 1007016245527451
315         rem = 0
```

```
316         for ii in numbers:
317             rem = (rem*16 + ii) % p
318         ret = ""
319         if rem==0:
320             ret = table[0]
321         while rem > 0:
322             ret += table[rem%64]
323             rem //= 64
324         return ret
325 
326     def clear(self):
327         r"""
328         Clears the saved data generated by this theory, and the cached
    functions.
329         """
330 
331         self._identify.cache_clear()
332         ns = "calcs/" + self._name + "."
333         for xx in os.listdir("calcs/"):
334             if xx.startswith(ns):
335                 os.remove("calcs/"+xx)
336 
337     def _save(self, ind, ret, is_table):
338         r"""
339         Saves the calculated data to persistent storage.
340         """
341         ns = "calcs/" + self._name + "."
342         if is_table:
343             excluded, n1, n2, large_ftype, ftype_inj = ind
344             numsind = [0]
345             for xx in excluded:
346                 numsind += xx.raw_numbers()
347             numsind += [n1, n2] + large_ftype.raw_numbers() + list(ftype_inj)
348         else:
349             excluded, n, ftype = ind
```

```
350            numsind = [1]
351            for xx in excluded:
352                numsind += xx.raw_numbers()
353            numsind += [n] + ftype.raw_numbers()
354        save_name = ns + self._compress(numsind)
355        os.makedirs(os.path.dirname(save_name), exist_ok=True)
356        with open(save_name, 'wb') as file:
357            pickle.dump(ret, file)
358
359    def _load(self, ind, is_table):
360        r"""
361        Tries to load persistent data.
362        """
363        ns = "calcs/" + self._name + "."
364
365        if is_table:
366            excluded, n1, n2, large_ftype, ftype_inj = ind
367            numsind = [0]
368            for xx in excluded:
369                numsind += xx.raw_numbers()
370            numsind += [n1, n2] + large_ftype.raw_numbers() + list(ftype_inj)
371        else:
372            excluded, n, ftype = ind
373            numsind = [1]
374            for xx in excluded:
375                numsind += xx.raw_numbers()
376            numsind += [n] + ftype.raw_numbers()
377        load_name = ns + self._compress(numsind)
378
379        if os.path.isfile(load_name):
380            with open(load_name, 'rb') as file:
381                ret = pickle.load(file)
382                if is_table:
383                    return ret
384                else:
```

```
385                     for xx in ret:
386                         xx._set_parent(self)
387                     return ret
388         return None
389
390     def _repr_(self):
391         r"""
392         Give a nice string representation of the theory object
393
394         OUTPUT: The representation as a string
395
396         EXAMPLES::
397
398             sage: from sage.algebras.flag_algebras import *
399             sage: print(GraphTheory)
400             Theory for Graph
401         """
402         return 'Theory for {}'.format(self._name)
403
404     def _element_constructor_(self, n, **kwds):
405         r"""
406         Construct elements of this theory
407
408         INPUT:
409
410         - ``n`` -- integer; number of points of the flag
411         - ``**kwds`` -- can contain ftype_points, listing
412             the points that will form part of the ftype;
413             and can contain the blocks for each signature.
414             If they are not included, they are assumed to
415             be empty lists.
416
417         OUTPUT: A Flag with the given parameters
418
419         EXAMPLES::
```

```
420
421         Create an empty graph on 3 vertices ::
422
423             sage: from sage.algebras.flag_algebras import *
424             sage: GraphTheory(3)
425             Flag on 3 points, ftype from [] with edges=[]
426
427         Create an edge with one point marked as an ftype ::
428
429             sage: GraphTheory(2, ftype_points=[0], edges=[[0, 1]])
430             Flag on 2 points, ftype from [0] with edges=[[0, 1]]
431
432         Create a RamseyGraphTheory flag, a fully colored
433         triangle (useful for calculating R(K_3), see
434         :func:`solve_problem`) ::
435
436             sage: RamseyGraphTheory(3, edges=[[0, 1], [0, 2], [1, 2]])
437             Flag on 3 points, ftype from [] with edges=[[0, 1], [0, 2], [1,
    2]], edges_marked=[]
438
439         .. NOTE::
440
441             Different input parameters can result in equal objects, for
442             example the following two graphs are automorphic::
443             sage: b1 = [[0, 1], [0, 2], [0, 4], [1, 3], [2, 4]]
444             sage: b2 = [[0, 4], [1, 2], [1, 3], [2, 3], [3, 4]]
445             sage: g1 = GraphTheory(5, edges=b1)
446             sage: g2 = GraphTheory(5, edges=b2)
447             sage: g1==g2
448             True
449
450         .. SEEALSO::
451
452             :func:`__init__` of :class:`Flag`
453         """
```

```
454	        return self.element_class(self, n, **kwds)
455
456	    def empty_element(self):
457	        r"""
458	        Returns the empty element, ``n``=0 and no blocks
459
460	        OUTPUT: The empty element of the CombinatorialTheory
461
462	        EXAMPLES::
463
464	            sage: from sage.algebras.flag_algebras import *
465	            sage: GraphTheory.empty_element()
466	            Ftype on 0 points with edges=[]
467
468	        .. NOTE::
469
470	            Since the underlying vertex set (empty set)
471	            is the same as the ftype point set, this is
472	            an ftype
473
474	        .. SEEALSO::
475
476	            :func:`empty`
477	        """
478	        return self.element_class(self, 0)
479
480	    empty = empty_element
481
482	    def _an_element_(self, n=0, ftype=None):
483	        r"""
484	        Returns a random element
485
486	        INPUT:
487
488	        - ``n`` -- integer (default: `0`); size of the element
```

```
489             - ``ftype`` -- Flag (default: `None`); ftype of the element
490                 if not provided then returns an element with empty ftype
491 
492             OUTPUT: A Flag with matching parameters
493             """
494             if ftype==None:
495                 ftype = self.empty_element()
496             if n==None or n==ftype.size():
497                 return ftype
498             ls = self.generate_flags(n, ftype)
499             return ls[randint(0, len(ls)-1)]
500 
501         def some_elements(self):
502             r"""
503             Returns a list of elements
504             """
505             pt = self.element_class(self, 1, ftype=[0])
506             return [self._an_element_(),
507                     self._an_element_(n=2),
508                     self._an_element_(ftype=pt),
509                     self._an_element_(n=2, ftype=pt)]
510 
511         def signature(self):
512             return self._signature
513 
514         def identify(self, n, ftype_points, **blocks):
515             r"""
516             The function used to test for equality.
517 
518             INPUT:
519 
520             - ``n`` -- integer; size of the flag
521             - ``ftype_points`` -- list; the points of the ftype
522             - ``**blocks`` -- the blocks for each signature
523 
```

```
524         OUTPUT: The identifier of the structure defined by the
525             ``identifier`` function in the __init__
526 
527         .. SEEALSO::
528 
529             :func:`Flag.unique`
530         """
531         blocks = {key:tuple([tuple(xx) for xx in blocks[key]])
532                   for key in blocks}
533         return self._identify(n, tuple(ftype_points), **blocks)
534 
535     @lru_cache(maxsize=None)
536     def _identify(self, n, ftype_points, **blocks):
537         r"""
538         The hidden _identify, the inputs are in a tuple form
539         and is cached for some speed
540         """
541         return self._identifier(n, ftype_points, **blocks)
542 
543     def exclude(self, flags=[]):
544         r"""
545         Exclude some induced flags from the theory
546 
547         This allows creation of CombinatorialTheory -s with excluded
548         flags. The flags are not allowed to appear as an induced
549         substructure in any of the generated flags later.
550 
551         INPUT:
552 
553         - ``flags`` -- list of flags or a flag (default: `[]`);
554             The list of flags to exclude, flags are treated as
555             a singleton list
556 
557         EXAMPLES::
558 
```

```
559        How to create triangle-free graphs ::
560
561            sage: from sage.algebras.flag_algebras import *
562            sage: triangle = GraphTheory(3, edges=[[0, 1], [0, 2], [1, 2]])
563            sage: GraphTheory.exclude(triangle)
564
565        There are 14 graphs on 5 vertices without triangles ::
566
567            sage: len(GraphTheory.generate_flags(5))
568            14
569
570        .. NOTE::
571
572            Calling :func:`exclude` again will overwrite the list
573            of excluded structures. So calling exclude() again, gives
574            back the original theory
575
576        TESTS::
577
578            sage: from sage.algebras.flag_algebras import *
579            sage: ThreeGraphTheory.exclude(ThreeGraphTheory(4))
580            sage: len(ThreeGraphTheory.generate_flags(5))
581            23
582            sage: TournamentTheory.exclude(TournamentTheory(3, edges=[[0, 1],
    [1, 2], [2, 0]]))
583            sage: TournamentTheory.generate_flags(5)
584            (Flag on 5 points, ftype from [] with edges=[[1, 0], [2, 0], [2,
    1], [3, 0], [3, 1], [3, 2], [4, 0], [4, 1], [4, 2], [4, 3]],)
585
586        """
587        if type(flags)==Flag:
588            self._excluded = [flags]
589        else:
590            self._excluded = flags
591
```

```
592     def _check_excluded(self, elms):
593         r"""
594         Helper to check the excluded structures in generation
595         """
596         flg = elms[0]
597         for xx in elms[1]:
598             if xx<= flg:
599                 return False
600         return True
601 
602     def optimize_problem(self, target_element, target_size, \
603                          ftypes=None, maximize=True, certificate=False, \
604                          positives=None):
605         r"""
606         Try to maximize or minimize the value of `target_element`
607 
608         The algorithm calculates the multiplication tables and
609         sends the SDP problem to cvxopt.
610 
611         INPUT:
612 
613         - ``target_element`` -- Flag or FlagAlgebraElement;
614             the target whose density this function tries to
615             maximize or minimize in large structures.
616         - ``target_size`` -- integer; The program evaluates
617             flags and the relations between them up to this
618             size.
619         - ``ftypes`` -- list of ftypes (default: `None`);
620             the list of ftypes the program will consider,
621             if not provided (or `None`) then generates all
622             possible ftypes.
623         - ``maximize`` -- boolean (default: `True`);
624         - ``certificate`` -- boolean (default: `False`);
625         - ``positives`` -- list of flag algebra elements,
626             optimizer will assume those are positive, can
```

```
627             have different types
628
629         OUTPUT: A bound for the optimization problem. If
630             certificate is requested then returns the entire
631             output of the solver as the second argument.
632
633         EXAMPLES::
634
635
636         Mantel's theorem, calculate the maximum density
637         of edges in triangle-free graphs ::
638
639             sage: from sage.algebras.flag_algebras import *
640             sage: GraphTheory.exclude(GraphTheory(3, edges=[[0, 1], [0, 2],
    [1, 2]]))
641             sage: x = GraphTheory.optimize_problem(GraphTheory(2, edges=[[0,
    1]]), 3)
642             ...
643             sage: abs(x-0.5)<1e-6
644             True
645
646         Generalized Turan problem, for example maximum density of K_3
647         in K_5 -free graphs. The complement is calculated, optimizing empty
648         E_3 in E_5 -free graphs. They are equivalent
649
650             sage: GraphTheory.exclude(GraphTheory(5))
651             sage: x = GraphTheory.optimize_problem(GraphTheory(3), 5) # long
    time (5 second)
652             ...
653
654
655         Generalized Turan problem for hypergraphs, maximum density of K^3_4
656         in K^3_5 -free 3-graphs. Again, the complement is calculated. This is
657         long and should not be normally tested
658
```

```
659             sage: ThreeGraphTheory.exclude(ThreeGraphTheory(5))
660             sage: ThreeGraphTheory.optimize_problem(ThreeGraphTheory(4), 6) #
    todo: not implemented
661             ...
662
663
664         The minimum number of transitive tournaments is attained at
665         a random tournament [CoRa2015]_::
666
667             sage: x = TournamentTheory.optimize_problem( \
668             ....: TournamentTheory(3, edges=[[0, 1], [0, 2], [1, 2]]), \
669             ....: 3, maximize=False)
670             ...
671
672
673         Ramsey's theorem, the K_5 is the largest 2-colorable complete
674         graph without monochromatic K_3. (see [LiPf2021]_) ::
675
676             sage: RamseyGraphTheory.exclude(RamseyGraphTheory(3, edges=[[0,
    1], [0, 2], [1, 2]]))
677             sage: x = RamseyGraphTheory.optimize_problem(RamseyGraphTheory(2),
     4, maximize=False)
678             ...
679             sage: abs(x-0.2)<1e-6
680             True
681
682         .. NOTE::
683
684             If `target_size` is too large, the calculation might take a
685             really long time. On a personal computer the following
686             maximum target_size values are recommended:
687             -GraphTheory: 8
688             -ThreeGraphTheory: 6
689             -DiGraphTheory: 5
690             -TournamentTheory: 9
```

```
691             -PermutationTheory: 8
692             -RamseyGraphTheory: 8
693             -OVGraphTheory: 5
694             -OEGraphTheory: 4
695         """
696         import time
697 
698         from csdpy import solve_sdp
699         import numpy as np
700         from tqdm import tqdm
701         import sys
702 
703         current = time.time()
704         #calculate constraints from positive vectors
705         if positives == None:
706             constraints_flags = []
707             constraints_vals = []
708         else:
709             constraints_flags = []
710             for ii in range(len(positives)):
711                 fv = positives[ii]
712                 if isinstance(fv, Flag):
713                     continue
714                 d = target_size - fv.size()
715                 k = fv.ftype().size()
716                 terms = fv.afae().parent().generate_flags(k+d)
717                 constraints_flags += [fv.mul_project(xx) for xx in terms]
718             constraints_vals = [0]*len(constraints_flags)
719 
720         #calculate ftypes
721         if ftypes is None:
722             flags = [flag for kk in range(2-target_size%2, target_size-1, 2)
723                          for flag in self.generate_flags(kk)]
724             ftypes = [flag.subflag([], ftype_points=list(range(flag.size())))
    \
```

```
725                       for flag in flags]
726 
727         print("Ftypes constructed in {:.2f}s".format(time.time() - current),
    flush=True); current = time.time()
728         block_sizes = [len(self.generate_flags((target_size + \
729                        ftype.size())//2, ftype)) for ftype in ftypes]
730         constraints = len(self.generate_flags(target_size))
731 
732         alg = FlagAlgebra(QQ, self)
733 
734         one_vector = target_element.ftype().project()<<(target_size -
    target_element.ftype().size())
735         constraints_flags.extend([one_vector, one_vector*(-1)])
736         constraints_vals.extend([1, -1])
737 
738         block_sizes.extend([-constraints, -2*len(constraints_vals)])
739         block_num = len(block_sizes)
740         mat_inds = []
741         mat_vals = []
742         print("Block sizes done in {:.2f}s".format(time.time() - current),
    flush=True); current = time.time()
743         print("Block sizes are {}".format(block_sizes), flush=True)
744         print("Calculating product matrices for {} ftypes and {} structures".
    format(len(ftypes), constraints), flush=True)
745         for ii, ftype in (pbar := tqdm(enumerate(ftypes), file=sys.stdout)):
746             ns = (target_size + ftype.size())//2
747             fls = self.generate_flags(ns, ftype)
748             table = self.mul_project_table(ns, ns, ftype, [])
749             for gg, mm in enumerate(table):
750                 dd = mm._dict()
751                 if len(dd)>0:
752                     inds, values = zip(*mm._dict().items())
753                     iinds, jinds = zip(*inds)
754                     for cc in range(len(iinds)):
755                         if iinds[cc]>=jinds[cc]:
```

```
756                             mat_inds.extend([gg+1, ii+1, iinds[cc]+1,
757                                              jinds[cc]+1])
758                             mat_vals.append(values[cc])
759             pbar.set_description("{} is complete".format(ftype))
760 
761         print("Table calculation done in {:.2f}s".format(time.time() - current
    ), flush=True); current = time.time()
762         if maximize:
763             avals = (target_element*(-1)<<(target_size - \
764                                                 target_element.size())).values()
765         else:
766             avals = (target_element<<(target_size - \
767                                           target_element.size())).values()
768 
769         for ii in range(len(constraints_vals)):
770             mat_inds.extend([0, block_num, 1+ii, 1+ii])
771             mat_vals.append(constraints_vals[ii])
772 
773         constraints_flags_vec = [(xx<<(target_size-xx.size())).values() for xx
     in constraints_flags]
774         for gg in range(constraints):
775             mat_inds.extend([gg+1, block_num-1, gg+1, gg+1])
776             mat_vals.append(1)
777             for ii in range(len(constraints_flags_vec)):
778                 mat_inds.extend([gg+1, block_num, ii+1, ii+1])
779                 mat_vals.append(constraints_flags_vec[ii][gg])
780         print("Target and constraint calculation done in {:.2f}s\n".format(
    time.time() - current), flush=True); current = time.time()
781 
782         sdp_result = solve_sdp(block_sizes, list(avals),
783                                mat_inds, mat_vals)
784         if maximize:
785             ret = max(-sdp_result['primal'], -sdp_result['dual'])
786         else:
787             ret = min(sdp_result['primal'], sdp_result['dual'])
```

```
788         print("Result is {}".format(ret), flush=True)
789         if certificate:
790             ralg = FlagAlgebra(RR, self)
791             vec = ralg(target_size, sdp_result['y'])
792             ret = (ret, vec, sdp_result)
793         return ret
794 
795     def _gfe(self, excluded, n, ftype):
796         r"""
797         Cached version of generate flags excluded
798 
799         .. SEEALSO::
800 
801             :func:`generate_flags`
802         """
803         if ftype==None:
804             ftype = self.empty()
805         ind = (excluded, n, ftype)
806         loaded = self._load(ind, False)
807         if loaded != None:
808             return loaded
809 
810         import multiprocessing as mp
811 
812         if ftype.size()==0: #just generate empty elements
813             if n==0:
814                 ret = (self.empty_element(), )
815                 self._save(ind, ret, False)
816                 return ret
817             if len(excluded)==0: #just return the output of the generator
818                 ret = tuple([self.element_class(self, n, **xx) for xx in self.
    _generator(n)])
819                 self._save(ind, ret, False)
820                 return ret
821 
```

```
822             #otherwise check each generated for the excluded values
823             slist = [(xx, excluded) for xx in self._gfe(tuple(), n, None)]
824             pool = mp.Pool(mp.cpu_count()-1)
825             canincl = pool.map(self._check_excluded, slist)
826             pool.close(); pool.join()
827             ret = tuple([slist[ii][0] for ii in range(len(slist)) if canincl[
    ii]])
828             self._save(ind, ret, False)
829             return ret
830 
831         #generate flags by first getting the empty structures then finding the
     flags
832         empstrs = self._gfe(excluded, n, None)
833         pool = mp.Pool(mp.cpu_count()-1)
834         pares = pool.map(ftype._ftypes_inside, empstrs)
835         pool.close(); pool.join()
836         ret = []
837         for coll in pares:
838             for xx in coll:
839                 if xx not in ret:
840                     ret.append(xx)
841         ret = tuple(ret)
842         self._save(ind, ret, False)
843         return ret
844 
845     def generate_flags(self, n, ftype=None):
846         r"""
847         Returns the list of flags with a given size and ftype
848 
849         INPUT:
850 
851         - ``n`` -- integer; the size of the returned structures
852         - ``ftype`` -- Flag; the ftype of the returned structures
853 
854         OUTPUT: List of all flags with given size and ftype
```

```
855
856         EXAMPLES::
857
858         There are 4 graphs on 3 vertices. Flags with empty
859         ftype correspond to elements of the theory ::
860
861             sage: from sage.algebras.flag_algebras import *
862             sage: len(GraphTheory.generate_flags(3))
863             4
864
865         There are 6 graph flags with one vertex ftype. The
866         "cherry" ([[0, 1], [0, 2]]) and the complement can be
867         marked two different ways to a flag
868
869             sage: len(GraphTheory.generate_flags(3, GraphTheory(1,
    ftype_points=[0])))
870             6
871
872         .. NOTE::
873
874             See the notes on :func:`optimize_problem`. A large `n` can
875             result in large number of structures.
876         """
877         if ftype==None:
878             ftype = self.empty()
879         else:
880             if not ftype.is_ftype():
881                 raise ValueError('{} is not an Ftype'.format(ftype))
882             if ftype not in self:
883                 raise ValueError('{} is not a part of this theory'.format(
    ftype))
884         ftype_size = ftype.size()
885         if n<ftype_size:
886             return tuple()
887         elif n==ftype_size:
```

```
888                 return (ftype, )
889             return self._gfe(tuple(self._excluded), n, ftype)
890 
891         def mul_project_table(self, n1, n2, large_ftype, ftype_inj=None):
892             r"""
893             Returns the multiplication projection table
894 
895             `ftype_inj` specifies a projection, an embedding of a
896             smaller ftype inside the `large_ftype`. Two flag vectors
897             with `n1` and `n2` underlying flag size set and `large_ftype`
898             ftype can be multiplied together and projected with this table.
899             The result is a tuple of sparse matrices corresponding to the
900             coefficients in the list of flags with the projected ftype
901 
902             INPUT:
903 
904             - ``n1`` -- integer; Vertex size of first flag vector
905             - ``n2`` -- integer; Vertex size of second flag vector
906             - ``large_ftype`` -- Flag; The ftype of the flag vectors
907             - ``ftype_inj`` -- list (default: `None`); The injection
908                 from the smaller ftype to the `large_ftype`. If `None`
909                 then the calculation is performed without any projection
910                 (so `ftype_inj` is the identity bijection)
911 
912             OUTPUT: A tuple of sparse matrices
913 
914             .. NOTE::
915 
916                 This just transforms the input to a standard form
917                 and then calls :func:`_mpte` which is cached for speed.
918                 This table is used in all sorts of operations, flag algebra
919                 multiplication and projection. But the main use happens in
920                 optimize_problem, where these tables form the semidefinite
921                 constraint.
922 
```

```
923        .. SEEALSO::
924
925            :func:`_mpte`
926            :func:`optimize_problem`
927            :func:`FlagAlgebraElement.mul_project`
928            :func:`FlagAlgebraElement._mul_`
929            :func:`FlagAlgebraElement.project`
930
931        TESTS::
932
933            sage: from sage.algebras.flag_algebras import *
934            sage: table = GraphTheory.mul_project_table(2, 2, GraphTheory(1,
    ftype_points=[0]), [])
935            sage: table[1][0, 0]
936            1/3
937
938            sage: table = RamseyGraphTheory.mul_project_table(3, 3,
    RamseyGraphTheory(2, ftype_points=[0, 1]), [])
939            sage: table[3][1, 1]
940            1/6
941        """
942        large_size = large_ftype.size()
943        if ftype_inj==None:
944            ftype_inj = tuple(range(large_size))
945        else:
946            ftype_inj = tuple(ftype_inj)
947            if len([ii for ii in ftype_inj if (ii not in range(large_size))])
    !=0:
948                raise ValueError('ftype_inj must map into the points of {}'.
    format(large_ftype))
949            if len(set(ftype_inj)) != len(ftype_inj):
950                raise ValueError('ftype_inj must be injective (no repeated
    elements)')
951        return self._mpte(tuple(self._excluded), n1, n2, large_ftype,
    ftype_inj)
```

```
952
953     def _mpte(self, excluded, n1, n2, large_ftype, ftype_inj):
954         r"""
955         The (hidden) cached version of :func:`mul_project_table`
956         """
957         ind = (excluded, n1, n2, large_ftype, ftype_inj)
958         loaded = self._load(ind, True)
959         if loaded != None:
960             return loaded
961
962         from sage.matrix.args import MatrixArgs
963         import multiprocessing as mp
964         ftype_inj = list(ftype_inj)
965         large_size = large_ftype.size()
966         small_ftype = large_ftype.subflag([], ftype_points=ftype_inj)
967         small_size = small_ftype.size()
968         ftype_remap = ftype_inj + [ii for ii in range(large_size) if (ii not
    in ftype_inj)]
969         N = n1 + n2 - large_size
970
971         Nflgs = self._gfe(excluded, N, small_ftype)
972         n1flgs = self._gfe(excluded, n1, large_ftype)
973         n2flgs = self._gfe(excluded, n2, large_ftype)
974
975         slist = tuple((flg, n1, n1flgs, n2, n2flgs, ftype_remap, large_ftype,
    small_ftype) for flg in Nflgs)
976
977         pool = mp.Pool(mp.cpu_count()-1)
978         mats = pool.map(self._density_wrapper, slist)
979         pool.close(); pool.join()
980
981         norm = falling_factorial(N - small_size, large_size - small_size)
982         norm *= binomial(N - large_size, n1 - large_size)
983
```

```
984         ret = tuple([MatrixArgs(QQ, mat[0], mat[1], entries=mat[2]).matrix()/
    norm for mat in mats])
985 
986         self._save(ind, ret, True)
987         return ret
988 
989     def _density_wrapper(self, ar):
990         r"""
991         Helper function used in the parallelization of calculating densities
992         """
993         return ar[0].densities(ar[1], ar[2], ar[3], ar[4], ar[5], ar[6], ar
    [7])
994 
995 class FlagAlgebraElement(CommutativeAlgebraElement):
996     def __init__(self, parent, n, values):
997         r"""
998         Initialize a Flag Algebra Element
999 
1000        INPUT:
1001 
1002        - ``parent`` -- FlagAlgebra; The parent :class:`FlagAlgebra`
1003        - ``n`` -- integer; the size of the flags
1004        - ``values`` -- vector; the values representing the
1005            linear combination of :class:`Flag` elements
1006 
1007        OUTPUT: A FlagAlgebraElement object with the given initial parameters
1008 
1009        .. NOTE::
1010 
1011            It is recommended to use the FlagAlgebra element constructor
1012 
1013        .. SEEALSO::
1014 
1015            :func:`FlagAlgebra._element_constructor_`
1016        """
```

```
1017         self._flags = parent.generate_flags(n)
1018         if len(values)!=len(self._flags):
1019             raise ValueError('The coefficients must have the same length as
     the number of flags')
1020         self._n = n
1021         base = parent.base()
1022         self._values = vector(base, values)
1023         self._ftype = parent.ftype()
1024         CommutativeAlgebraElement.__init__(self, parent)
1025 
1026     def ftype(self):
1027         r"""
1028         Return the ftype of the parent FlagAlgebra
1029 
1030         The algebra is defined over elements of a CombinatorialTheory, all
1031         having the same ftype.
1032 
1033         OUTPUT: The ftype of the parent FlagAlgebra. A :class:`Flag` element
1034 
1035         EXAMPLES::
1036 
1037         The ftype of a :class:`Flag` is the same as the ftype of the
1038         :class:`FlagAlgebraElement` we can construct from it ::
1039 
1040             sage: from sage.algebras.flag_algebras import *
1041             sage: g = GraphTheory(3)
1042             sage: g.ftype()
1043             Ftype on 0 points with edges=[]
1044             sage: g.ftype()==g.afae().ftype()
1045             True
1046 
1047         .. SEEALSO::
1048 
1049             :func:`ftype` in :class:`FlagAlgebra`
1050             :func:`ftype` in :class:`Flag`
```

```
1051         """
1052         return self._ftype
1053 
1054     def size(self):
1055         r"""
1056         Return the size of the vertex set of flags in this element
1057 
1058         OUTPUT: The size of each flag is :func:`flags`.
1059 
1060         TESTS::
1061 
1062             sage: from sage.algebras.flag_algebras import *
1063             sage: FG = FlagAlgebra(QQ, GraphTheory)
1064             sage: FGElem = FG._an_element_()
1065             sage: FGElem.size() == FGElem.flags()[0].size()
1066             True
1067         """
1068         return self._n
1069 
1070     vertex_number = size
1071 
1072     def flags(self):
1073         r"""
1074         Returns the list of flags, corresponding to each base element
1075 
1076         The flags returned are the list of flags with size equal to
1077         `self.size()` and ftype to `self.ftype()`. Their number is
1078         the same as the length of `self.values()`.
1079 
1080         OUTPUT: The list of flags
1081 
1082         EXAMPLES::
1083 
1084         3 vertex graphs with empty ftype ::
1085 
```

```
1086            sage: from sage.algebras.flag_algebras import *
1087            sage: g = GraphTheory(3)
1088            sage: g.afae()
1089            Flag Algebra Element over Rational Field
1090            1 - Flag on 3 points, ftype from [] with edges=[]
1091            0 - Flag on 3 points, ftype from [] with edges=[[0, 2]]
1092            0 - Flag on 3 points, ftype from [] with edges=[[0, 2], [1, 2]]
1093            0 - Flag on 3 points, ftype from [] with edges=[[0, 1], [0, 2],
    [1, 2]]
1094            sage: g.afae().flags()
1095            (Flag on 3 points, ftype from [] with edges=[],
1096             Flag on 3 points, ftype from [] with edges=[[0, 2]],
1097             Flag on 3 points, ftype from [] with edges=[[0, 2], [1, 2]],
1098             Flag on 3 points, ftype from [] with edges=[[0, 1], [0, 2], [1,
    2]])
1099
1100        .. NOTE::
1101
1102            This is the same as
1103            `self.theory().generate_flags(self.size(), self.ftype())`
1104
1105        .. SEEALSO::
1106
1107            :func:`CombinatorialTheory.generate_flags`
1108            :func:`size`
1109            :func:`ftype`
1110            :func:`values`
1111            :func:`Flag.afae`
1112
1113        TESTS::
1114
1115            sage: from sage.algebras.flag_algebras import *
1116            sage: g.afae().flags() == g.theory().generate_flags(g.size(), g.
    ftype())
1117            True
```

```
1118        """
1119        return self._flags
1120
1121    def values(self):
1122        r"""
1123        Returns the vector of values, corresponding to each element
1124        in :func:`flags`
1125
1126        OUTPUT: A vector
1127
1128        EXAMPLES::
1129
1130        A flag transformed to a flag algebra element has
1131        all zeroes except one entry, itself ::
1132
1133            sage: from sage.algebras.flag_algebras import *
1134            sage: g = GraphTheory(3)
1135            sage: g.afae().values()
1136            (1, 0, 0, 0)
1137
1138        .. SEEALSO::
1139
1140            :func:`flags`
1141            :func:`_vector_`
1142            :func:`Flag.afae`
1143        """
1144        return self._values
1145
1146    def _vector_(self, R):
1147        r"""
1148        Returns the vector of values, corresponding to each element
1149        in :func:`flags` in a given base.
1150
1151        OUTPUT: A vector
1152
```

```
1153         EXAMPLES::
1154 
1155         A flag transformed to a flag algebra element has
1156         all zeroes except one entry, itself ::
1157 
1158             sage: from sage.algebras.flag_algebras import *
1159             sage: g = GraphTheory(3)
1160             sage: g.afae()._vector_(QQ['x'])
1161             (1, 0, 0, 0)
1162 
1163         .. SEEALSO::
1164 
1165             :func:`values()`
1166             :func:`Flag.afae`
1167         """
1168         return vector(R, self._values)
1169 
1170     def __len__(self):
1171         r"""
1172         Returns the length, the number of elements
1173         in :func:`flags` or :func:`values` (which is the same)
1174 
1175         EXAMPLES::
1176 
1177             sage: from sage.algebras.flag_algebras import *
1178             sage: g = GraphTheory(3)
1179             sage: len(g.afae())
1180             4
1181 
1182         .. SEEALSO::
1183 
1184             :func:`flags`
1185             :func:`values`
1186             :func:`Flag.afae`
1187             :func:`__iter__`
```

```
1188            """
1189            return len(self._values)
1190
1191        def __iter__(self):
1192            r"""
1193            Iterates over the elements of self
1194
1195            It yields (number, flag) tuples, indicating the
1196            coefficient of the flag.
1197
1198            EXAMPLES::
1199
1200                sage: from sage.algebras.flag_algebras import *
1201                sage: g = GraphTheory(3)
1202                sage: for x in g.afae():
1203                ....:     print(x)
1204                (1, Flag on 3 points, ftype from [] with edges=[])
1205                (0, Flag on 3 points, ftype from [] with edges=[[0, 2]])
1206                (0, Flag on 3 points, ftype from [] with edges=[[0, 2], [1, 2]])
1207                (0, Flag on 3 points, ftype from [] with edges=[[0, 1], [0, 2],
        [1, 2]])
1208
1209
1210            .. SEEALSO::
1211
1212                :func:`flags`
1213                :func:`values`
1214                :func:`Flag.afae`
1215                :func:`__len__`
1216            """
1217            for ii in range(len(self)):
1218                yield (self._values[ii], self._flags[ii])
1219
1220        def _repr_(self):
1221            r"""
```

```
1222        Give a string representation
1223
1224        Lists the flags and the corresponding coefficients,
1225        each on a separate line. If the list is too long
1226        then only shows nonzero entries.
1227
1228
1229        EXAMPLES::
1230
1231        Short list, so display all ::
1232
1233            sage: from sage.algebras.flag_algebras import *
1234            sage: gf = GraphTheory(3).afae()
1235            sage: gf
1236            Flag Algebra Element over Rational Field
1237            1 - Flag on 3 points, ftype from [] with edges=[]
1238            0 - Flag on 3 points, ftype from [] with edges=[[0, 2]]
1239            0 - Flag on 3 points, ftype from [] with edges=[[0, 2], [1, 2]]
1240            0 - Flag on 3 points, ftype from [] with edges=[[0, 1], [0, 2],
    [1, 2]]
1241
1242        Long list, only the nonzero entries are displayed
1243        (for some reason this is failing the doctest, but should show
1244        `Flag Algebra Element over Rational Field
1245        1 - Flag on 5 points, ftype from [] with edges=[]
1246        1 - Flag on 5 points, ftype from [] with edges=[[0, 3], [1, 4]]
1247
1248        ` I can't figure out why)::
1249
1250            sage: g1 = GraphTheory(5)
1251            sage: g2 = GraphTheory(5, edges=[[0, 1], [3, 4]])
1252            sage: g1+g2
1253            ...
1254
1255        .. SEEALSO::
```

```
1256
1257            :func:`Flag._repr_`
1258        """
1259        sttrl = ['Flag Algebra Element over {}'.format(self.parent().base())]
1260        strs = [str(xx) for xx in self.values()]
1261        maxstrlen = max([len(xx) for xx in strs])
1262        flgs = self.flags()
1263        for ii in range(len(self)):
1264            if len(self)<20:
1265                sttrl.append(('{:<'+str(maxstrlen)+'} - {}').format(strs[ii],
    str(flgs[ii])))
1266            else:
1267                include = True
1268                try:
1269                    include = abs(float(self.values()[ii]))>=1e-8
1270                except:
1271                    include = self.values()[ii]!=0
1272                if include:
1273                    sttrl.append(('{:<'+str(maxstrlen)+'} - {}').format(strs[
    ii], str(flgs[ii])))
1274        return "\n".join(sttrl)
1275
1276    def as_flag_algebra_element(self):
1277        r"""
1278        Returns self.
1279
1280        Only here to allow calling this function on
1281        both flags and flag algebra elements
1282
1283        .. SEEALSO::
1284
1285            :func:`Flag.afae`
1286        """
1287        return self
1288
```

```
1289     afae = as_flag_algebra_element
1290
1291     def _add_(self, other):
1292         r"""
1293         Adds to FlagAlgebraElements together
1294
1295         OUTPUT: The sum
1296
1297         EXAMPLES::
1298
1299         The smaller size is shifted to match the larger ::
1300
1301             sage: from sage.algebras.flag_algebras import *
1302             sage: g = GraphTheory(3).afae()
1303             sage: e = GraphTheory(2).afae()
1304             sage: e+g
1305             Flag Algebra Element over Rational Field
1306             2   - Flag on 3 points, ftype from [] with edges=[]
1307             2/3 - Flag on 3 points, ftype from [] with edges=[[0, 2]]
1308             1/3 - Flag on 3 points, ftype from [] with edges=[[0, 2], [1, 2]]
1309             0   - Flag on 3 points, ftype from [] with edges=[[0, 1], [0, 2],
     [1, 2]]
1310
1311         .. NOTE::
1312
1313             The result's size will match the size of the larger component
1314
1315         .. SEEALSO::
1316
1317             :func:`Flag._add_`
1318             :func:`__lshift__`
1319             :func:`_sub_`
1320         """
1321         nm = max(self.size(), other.size())
```

```
1322         vals = (self<<(nm-self.size())).values() + (other<<(nm-other.size())).
    values()
1323         return self.__class__(self.parent(), nm, vals)
1324 
1325     def _sub_(self, other):
1326         r"""
1327         Subtract a FlagAlgebraElement from this
1328 
1329         EXAMPLES::
1330 
1331         This also shifts the smaller flag to match the larger ::
1332 
1333             sage: from sage.algebras.flag_algebras import *
1334             sage: g = GraphTheory(3).afae()
1335             sage: e = GraphTheory(2).afae()
1336             sage: e-g
1337             Flag Algebra Element over Rational Field
1338             0   - Flag on 3 points, ftype from [] with edges=[]
1339             2/3 - Flag on 3 points, ftype from [] with edges=[[0, 2]]
1340             1/3 - Flag on 3 points, ftype from [] with edges=[[0, 2], [1, 2]]
1341             0   - Flag on 3 points, ftype from [] with edges=[[0, 1], [0, 2],
    [1, 2]]
1342 
1343         .. SEEALSO::
1344 
1345             :func:`Flag._sub_`
1346             :func:`__lshift__`
1347             :func:`_add_`
1348         """
1349         nm = max(self.size(), other.size())
1350         vals = (self<<(nm-self.size())).values() - (other<<(nm-other.size())).
    values()
1351         return self.__class__(self.parent(), nm, vals)
1352 
1353     def _mul_(self, other):
```

```
1354        r"""
1355        Multiplies two elements together
1356
1357        The result will have size
1358        `self.size() + other.size() - self.ftype().size()`
1359
1360        EXAMPLES::
1361
1362        Two empty edges multiplied together has size 4 ::
1363
1364            sage: from sage.algebras.flag_algebras import *
1365            sage: e = GraphTheory(2).afae()
1366            sage: (e*e).size()
1367            4
1368
1369        But if pointed (size of ftype is 1), then the size is 3 ::
1370
1371            sage: pe = GraphTheory(2, ftype=[0])
1372            sage: pe*pe
1373            Flag Algebra Element over Rational Field
1374            1 - Flag on 3 points, ftype from [0] with edges=[]
1375            0 - Flag on 3 points, ftype from [0] with edges=[[0, 2]]
1376            1 - Flag on 3 points, ftype from [1] with edges=[[0, 2]]
1377            0 - Flag on 3 points, ftype from [0] with edges=[[0, 2], [1, 2]]
1378            0 - Flag on 3 points, ftype from [2] with edges=[[0, 2], [1, 2]]
1379            0 - Flag on 3 points, ftype from [0] with edges=[[0, 1], [0, 2],
    [1, 2]]
1380
1381        Can also multiply with constants:
1382
1383            sage: pe*3
1384            Flag Algebra Element over Rational Field
1385            3 - Flag on 2 points, ftype from [0] with edges=[]
1386            0 - Flag on 2 points, ftype from [0] with edges=[[0, 1]]
1387
```

```
1388        .. NOTE::
1389
1390            Multiplying and then projecting to a smaller ftype
1391            can be performed more efficiently with :func:`mul_project`
1392
1393        .. SEEALSO::
1394
1395            :func:`Flag._mul_`
1396            :func:`mul_project`
1397        """
1398        if self.size()==self.ftype().size():
1399            vals = other.values() * self.values()[0]
1400            return self.__class__(self.parent(), other.size(), vals)
1401        if other.size()==self.ftype().size():
1402            vals = self.values() * other.values()[0]
1403            return self.__class__(self.parent(), self.size(), vals)
1404        table = self.parent().mpt(self.size(), other.size())
1405        N = self.size() + other.size() - self.ftype().size()
1406        vals = [self.values() * mat * other.values() for mat in table]
1407        return self.__class__(self.parent(), N, vals)
1408
1409    def __truediv__(self, other):
1410        r"""
1411        Divide by a scalar
1412
1413        INPUT:
1414
1415        - ``other`` -- number; any number such that `1` can be divided with
    that
1416
1417        OUTPUT: The `FlagAlgebraElement` resulting from the division
1418
1419        EXAMPLES::
1420
1421        If 1 can be divided by that, then the division is allowed ::
```

```
1422
1423            sage: from sage.algebras.flag_algebras import *
1424            sage: var('x')
1425            x
1426            sage: g = GraphTheory(2)
1427            sage: g.afae()/x
1428            Flag Algebra Element over Symbolic Ring
1429            1/x - Flag on 2 points, ftype from [] with edges=[]
1430            0   - Flag on 2 points, ftype from [] with edges=[[0, 1]]
1431
1432        .. NOTE::
1433
1434            This is the linear extension of :func:`Flag.__truediv__`
1435
1436        .. SEEALSO::
1437
1438            :func:`Flag.afae`
1439            :func:`Flag.__truediv__`
1440        """
1441        return self * (1/other)
1442
1443    def __lshift__(self, amount):
1444        r"""
1445        `FlagAlgebraElement`, equal to this, with size is shifted by the
    amount
1446
1447        The result will have size equal to
1448        `self.size() + amount`, but the elements will be equal
1449
1450        EXAMPLES::
1451
1452        Edge shifted to size `3` ::
1453
1454            sage: from sage.algebras.flag_algebras import *
1455            sage: edge = GraphTheory(2, edges=[[0, 1]])
```

```
1456             sage: (edge.afae()<<1).values()
1457             (0, 1/3, 2/3, 1)
1458 
1459         .. NOTE::
1460 
1461             This is the linear extension of :func:`Flag.__lshift__`
1462 
1463         .. SEEALSO::
1464 
1465             :func:`Flag.__lshift__`
1466             :func:`Flag.afae()`
1467         """
1468         if amount<0:
1469             raise ValueError('Can not have negative shifts')
1470         if amount==0:
1471             return self
1472         table = self.parent().mpt(self.size(), amount + self.ftype().size())
1473         vals = [sum(self.values() * mat) for mat in table]
1474         return self.__class__(self.parent(), self.size() + amount, vals)
1475 
1476     def __getitem__(self, flag):
1477         if (not isinstance(flag, Flag)) or (not flag.ftype()==self.ftype()) or
     (not self.size()==flag.size()):
1478             raise TypeError("Indecies must be Flags with matching ftype and
    size, not {}".format(str(type(flag))))
1479         return self.values()[self.flags().index(flag)]
1480 
1481     def __setitem__(self, flag, value):
1482         if (not isinstance(flag, Flag)) or (not flag.ftype()==self.ftype()) or
     (not self.size()==flag.size()):
1483             raise TypeError("Indecies must be Flags with matching ftype and
    size, not {}".format(str(type(flag))))
1484         self.values()[self.flags().index(flag)] = value
1485 
1486     def project(self, ftype_inj=tuple()):
```

```
1487        r"""
1488        Project this to a smaller ftype
1489
1490
1491        INPUT:
1492
1493        - ``ftype_inj`` -- tuple (default: (, )); the injection of the
1494            projected ftype inside the larger ftype
1495
1496        OUTPUT: the `FlagAlgebraElement` resulting from the projection
1497
1498        EXAMPLES::
1499
1500        If the center of a cherry is flagged, then the projection has
1501        coefficient 1/3 ::
1502
1503            sage: from sage.algebras.flag_algebras import *
1504            sage: p_cherry = GraphTheory(3, edges=[[0, 1], [0, 2]],
    ftype_points=[0])
1505            sage: p_cherry.afae().project().values()
1506            (0, 0, 1/3, 0)
1507
1508        .. NOTE::
1509
1510            This is the linear extension of :func:`Flag.project`
1511
1512            If `ftype_inj==tuple(range(self.ftype().size()))` then this
1513            does nothing.
1514
1515        .. SEEALSO::
1516
1517            :func:`Flag.project`
1518        """
1519        return self.mul_project(1, ftype_inj)
1520
```

```
1521    def mul_project(self, other, ftype_inj=tuple()):
1522        r"""
1523        Multiply self with other, and the project the result.
1524
1525        INPUT:
1526
1527        - ``ftype_inj`` -- tuple (default: (, )); the injection of the
1528            projected ftype inside the larger ftype
1529
1530        OUTPUT: the `FlagAlgebraElement` resulting from the multiplication
1531            and projection
1532
1533        EXAMPLES::
1534
1535        Pointed edge multiplied with itself and projected ::
1536
1537            sage: from sage.algebras.flag_algebras import *
1538            sage: felem = GraphTheory(2, edges=[[0, 1]], ftype_points=[0]).
    afae()
1539            sage: felem.mul_project(felem).values()
1540            (0, 0, 1/3, 1)
1541
1542        .. NOTE::
1543
1544            This is the bilinear extension of :func:`Flag.mul_project`
1545
1546            If `ftype_inj==tuple(range(self.ftype().size()))` then this
1547            is the same as usual multiplication.
1548
1549        .. SEEALSO::
1550
1551            :func:`_mul_`
1552            :func:`project`
1553            :func:`Flag.mul_project`
1554        """
```

```
1555        other = self.parent(other)
1556        ftype_inj = tuple(ftype_inj)
1557        new_ftype = self.ftype().subflag([], ftype_points=ftype_inj)
1558        table = self.parent().mpt(self.size(), other.size(), ftype_inj)
1559        N = self.size() + other.size() - self.ftype().size()
1560        vals = [self.values() * mat * other.values() for mat in table]
1561
1562        TargetAlgebra = FlagAlgebra(self.parent().base(), self.parent().
    combinatorial_theory(), new_ftype)
1563        return TargetAlgebra(N, vals)
1564
1565    def density(self, other):
1566        r"""
1567        The density of self in other.
1568
1569        Randomly choosing self.size() points in other, the
1570        probability of getting self.
1571
1572        EXAMPLES::
1573
1574        Density of an edge in the cherry graph is 2/3 ::
1575
1576            sage: from sage.algebras.flag_algebras import *
1577            sage: cherry = GraphTheory(3, edges=[[0, 1], [0, 2]]).afae()
1578            sage: edge = GraphTheory(2, edges=[[0, 1]]).afae()
1579            sage: cherry.density(edge)
1580            2/3
1581
1582        .. NOTE::
1583
1584            This is the bilinear extension of :func:`Flag.mul_project`
1585
1586        .. SEEALSO::
1587
1588            :func:`Flag.density`
```

```
1589         """
1590         diff = self.size() - other.size()
1591         if diff < 0:
1592             raise ValueError('The target can not be larger')
1593         return self.values() * (other<<diff).values()
1594 
1595     def _richcmp_(self, other, op):
1596         r"""
1597         Compares the elements.
1598 
1599         Since the parent agrees, the ftype too. They are shifted to the
1600         same size and the values compared elementwise.
1601 
1602         EXAMPLES::
1603 
1604         Trivial example `g <= 2*g` ::
1605 
1606             sage: from sage.algebras.flag_algebras import *
1607             sage: g = GraphTheory(3).afae()
1608             sage: g <= 2*g
1609             True
1610 
1611         Since `g` has zero coefficients ::
1612 
1613             sage: g < 2*g
1614             False
1615 
1616         Shifting the size gives equal elements ::
1617 
1618             sage: g == (g<<1)
1619             True
1620 
1621         More sophisticated example, Mantel's theorem.
1622         Using the fact that for large enough structures
1623         `x.mul_project(x)` is always positive, we get that
```

```
1624        `1/2 >= GraphTheory(2)`, so K_3 free graphs can not
1625        contain more than 1/2 density of K_2 ::
1626
1627            sage: GraphTheory.exclude(GraphTheory(3))
1628            sage: f = GraphTheory(2, ftype=[0]) - 1/2
1629            sage: 1/2 >= f.mul_project(f)*2 + GraphTheory(2)
1630            True
1631
1632        .. NOTE::
1633
1634            When comparing Flags with FlagAlgebraElements, it
1635            will return False, unless the Flag is transformed
1636            to a FlagAlgebraElement.
1637
1638        .. SEEALSO::
1639
1640            :func:`mul_project`
1641        """
1642        nm = max(self.size(), other.size())
1643        v1 = (self<<(nm-self.size())).values()
1644        v2 = (other<<(nm-other.size())).values()
1645        return all([richcmp(v1[ii], v2[ii], op) for ii in range(len(v1))])
1646
1647 class FlagAlgebra(CommutativeAlgebra, UniqueRepresentation):
1648     def __init__(self, base, theory, ftype=None):
1649         r"""
1650         Initialize a FlagAlgebra
1651
1652         INPUT:
1653
1654         - ``base`` -- Ring; The base ring, this FlagAlgebra is constructed
     over
1655             This must contain the rationals `QQ`
1656         - ``theory`` -- CombinatorialTheory; The combinatorial theory
1657             this flag algebra is based on
```

```
1658        - ``ftype`` -- Flag (default=`None`); The ftype of the elements
1659            in this FlagAlgebra. The default `None` gives the empty type
1660
1661        OUTPUT: The resulting FlagAlgebra
1662
1663        EXAMPLES::
1664
1665        Create the FlagAlgebra for GraphTheory (without any ftype) ::
1666
1667            sage: from sage.algebras.flag_algebras import *
1668            sage: GraphFlagAlgebra = FlagAlgebra(QQ, GraphTheory)
1669            sage: GraphFlagAlgebra
1670            Flag Algebra with Ftype on 0 points with edges=[] over Rational
    Field
1671
1672        Create the FlagAlgebra for TournamentTheory with point ftype ::
1673
1674            sage: FlagAlgebra(QQ, TournamentTheory, TournamentTheory(1,
    ftype_points=[0]))
1675            Flag Algebra with Ftype on 1 points with edges=[] over Rational
    Field
1676        """
1677        if ftype==None:
1678            ftype = theory.empty_element()
1679        else:
1680            if not ftype.is_ftype():
1681                raise ValueError('{} is not an Ftype'.format(ftype))
1682            if ftype.theory() != theory:
1683                raise ValueError('{} is not a part of {}'.format(ftype, theory
    ))
1684        if not base.has_coerce_map_from(QQ):
1685            raise ValueError('The base must contain the rationals')
1686
1687        self._theory = theory
1688        self._ftype = ftype
```

```
1689            CommutativeAlgebra.__init__(self, base)
1690
1691        Element = FlagAlgebraElement
1692
1693        def _element_constructor_(self, *args, **kwds):
1694            r"""
1695            Constructs a FlagAlgebraElement with the given parameters
1696
1697            If a single value is provided it constructs the constant in the
        Algebra
1698            If a Flag is provided it constructs the element whose only `1` value
        is
1699            that Flag.
1700            If a different FlagAlgebraElement is provided, then checks if the
1701            `theory` and the `ftype` agrees, then tries to coerce the values to
        the
1702            base of self.
1703            Otherwise uses the constructor of FlagAlgebraElement, which accepts a
1704            size value and a list of coefficients, whose length must be precisely
        the
1705            number of flags.
1706
1707            EXAMPLES::
1708
1709            Construct from a constant ::
1710
1711                sage: from sage.algebras.flag_algebras import *
1712                sage: FA = FlagAlgebra(QQ, GraphTheory)
1713                sage: FA(3)
1714                Flag Algebra Element over Rational Field
1715                3 - Ftype on 0 points with edges=[]
1716
1717            Construct from a flag ::
1718
1719                sage: g = GraphTheory(2)
```

```
1720            sage: el = FA(g)
1721            sage: el
1722            Flag Algebra Element over Rational Field
1723            1 - Flag on 2 points, ftype from [] with edges=[]
1724            0 - Flag on 2 points, ftype from [] with edges=[[0, 1]]
1725
1726        Construct from a FlagAlgebraElement with smaller base ::
1727
1728            sage: FAX = FlagAlgebra(QQ['x'], GraphTheory)
1729            sage: FAX(el)
1730            Flag Algebra Element over Univariate Polynomial Ring in x over
    Rational Field
1731            1 - Flag on 2 points, ftype from [] with edges=[]
1732            0 - Flag on 2 points, ftype from [] with edges=[[0, 1]]
1733
1734        Constructing the element directly from coefficients ::
1735
1736            sage: FA(2, [3, 4])
1737            Flag Algebra Element over Rational Field
1738            3 - Flag on 2 points, ftype from [] with edges=[]
1739            4 - Flag on 2 points, ftype from [] with edges=[[0, 1]]
1740
1741        .. SEEALSO::
1742
1743            :func:`FlagAlgebraElement.__init__`
1744        """
1745        if len(args)==1:
1746            v = args[0]
1747            base = self.base()
1748            if isinstance(v, Flag):
1749                if v.ftype()==self.ftype():
1750                    flags = self.generate_flags(v.size())
1751                    vec = vector(base, [(1 if xx==v else 0) for xx in flags])
1752                    return self.element_class(self, v.size(), vec)
1753            elif isinstance(v, FlagAlgebraElement):
```

```
1754                if v.ftype()==self.ftype():
1755                    if self.base()==v.parent().base():
1756                        return v
1757                    elif self.base().has_coerce_map_from(v.parent().base()):
1758                        vals = vector(self.base(), v.values())
1759                        return self.element_class(self, v.size(), vals)
1760            elif v in base:
1761                return self.element_class(self, self.ftype().size(), vector(
    base, [v]))
1762            raise ValueError('Can\'t construct an element from {}'.format(v))
1763        return self.element_class(self, *args, **kwds)
1764
1765    def _coerce_map_from_(self, S):
1766        r"""
1767        Checks if it can be coerced from S
1768        """
1769        if self.base().has_coerce_map_from(S):
1770            return True
1771        if S==self.theory():
1772            return True
1773        if isinstance(S, FlagAlgebra):
1774            if S.ftype()==self.ftype() and self.base().has_coerce_map_from(S.
    base()):
1775                return True
1776        return False
1777
1778    def _pushout_(self, S):
1779        r"""
1780        Constructs the pushout FlagAlgebra
1781        """
1782        if S.has_coerce_map_from(self.base()):
1783            return FlagAlgebra(S, self.theory(), self.ftype())
1784        return None
1785
1786    def _repr_(self):
```

```
1787        r"""
1788        Returns a short text representation
1789
1790        EXAMPLES::
1791
1792            sage: from sage.algebras.flag_algebras import *
1793            sage: FlagAlgebra(QQ, GraphTheory)
1794            Flag Algebra with Ftype on 0 points with edges=[] over Rational
    Field
1795
1796        .. SEEALSO::
1797
1798            :func:`Flag._repr_`
1799        """
1800        return 'Flag Algebra with {} over {}'.format(self.ftype(), self.base()
    )
1801
1802    def ftype(self):
1803        r"""
1804        Returns the ftype of this FlagAlgebra.
1805
1806        EXAMPLES::
1807
1808        Without specifying anything in the constructor, the ftype
1809        is empty ::
1810
1811            sage: from sage.algebras.flag_algebras import *
1812            sage: FA = FlagAlgebra(QQ, GraphTheory)
1813            sage: FA.ftype()
1814            Ftype on 0 points with edges=[]
1815
1816        .. NOTE::
1817
1818            This is the same ftype as the ftype of the elements,
1819            and the ftype of the flags in those elements.
```

```
1820
1821         .. SEEALSO::
1822
1823             :func:`Flag.ftype`
1824             :func:`FlagAlgebraElement.ftype`
1825         """
1826         return self._ftype
1827
1828     def combinatorial_theory(self):
1829         r"""
1830         Returns the :class:`CombinatorialTheory` object, whose
1831         flags form the basis of this FlagAlgebra
1832
1833         EXAMPLES::
1834
1835         This is the same as provided in the constructor ::
1836
1837             sage: from sage.algebras.flag_algebras import *
1838             sage: FA = FlagAlgebra(QQ, GraphTheory)
1839             sage: FA.theory()
1840             Theory for Graph
1841
1842         .. SEEALSO::
1843
1844             :func:`__init__`
1845             :class:`CombinatorialTheory`
1846         """
1847         return self._theory
1848
1849     theory = combinatorial_theory
1850
1851     def base_ring(self):
1852         r"""
1853         Returns the base_ring
1854
```

```
1855        Same as `base_ring` of the `base` provided in the constructor
1856
1857        EXAMPLES::
1858
1859            sage: from sage.algebras.flag_algebras import *
1860            sage: FA = FlagAlgebra(QQ, GraphTheory)
1861            sage: FA.base_ring()
1862            Rational Field
1863
1864        .. SEEALSO::
1865
1866            :func:`base`
1867        """
1868        return self.base().base_ring()
1869
1870    def characteristic(self):
1871        r"""
1872        Returns the characteristic
1873
1874        Same as `characteristic` of the `base` provided in the constructor
1875
1876        EXAMPLES::
1877
1878            sage: from sage.algebras.flag_algebras import *
1879            sage: FA = FlagAlgebra(QQ, GraphTheory)
1880            sage: FA.characteristic()
1881            0
1882
1883        .. SEEALSO::
1884
1885            :func:`base`
1886        """
1887        return self.base().characteristic()
1888
1889    def generate_flags(self, n):
```

```
1890        r"""
1891        Generates flags of a given size, and `ftype` of `self`
1892
1893        .. NOTE::
1894
1895            Same as `CombinatorialTheory.generate_flags` with `ftype`
1896            from `self`
1897
1898        .. SEEALSO::
1899
1900            :func:`CombinatorialTheory.generate_flags`
1901        """
1902        return self.theory().generate_flags(n, self.ftype())
1903
1904    def _an_element_(self):
1905        r"""
1906        Returns an element
1907        """
1908        a = self.base().an_element()
1909        f = self.combinatorial_theory()._an_element_(n=self.ftype().size() +
     1, ftype=self.ftype())
1910        return self(f)*a
1911
1912    def some_elements(self):
1913        r"""
1914        Returns a small list of elements
1915        """
1916        return [self.an_element(),self(self.base().an_element())]
1917
1918    def mul_project_table(self, n1, n2, ftype_inj=None):
1919        r"""
1920        Returns the multiplication projection table
1921
1922        This is the same as :func:`CombinatorialTheory.mul_project_table`
1923        with self.ftype() substituted in.
```

```
1924
1925        .. SEEALSO::
1926
1927            :func:`CombinatorialTheory.mul_project_table`
1928        """
1929        return self.theory().mul_project_table(n1, n2, self.ftype(), ftype_inj
    )
1930
1931    mpt = mul_project_table
1932
1933
1934 #################################################
1935 #
1936 #    Implementation of a few (common) theories
1937 #
1938 #################################################
1939
1940
1941 def _generator_graph(n):
1942     r"""
1943     Given `n` integer, generates the graphs of size `n` using nauty
1944     and returns them in a dictionary required for Flag constructors
1945     """
1946     for xx in graphs.nauty_geng(str(n)):
1947         yield {'edges': tuple(xx.edges(labels=None))}
1948
1949 def _identify_graph(n, ftype_points, edges):
1950     r"""
1951     Creates a unique identifier for a graph using canonical labelings
1952     """
1953     partition = [[ii] for ii in ftype_points] + [list(set(range(n)).difference
    (set(ftype_points))), ]
1954     g = Graph([list(range(n)), edges], format='vertices_and_edges')
1955     blocks = tuple(g.canonical_label(partition=partition).edges(labels=None,
    sort=True))
```

```
1956     ftype_points = tuple(range(len(ftype_points)))
1957     return (n, ftype_points, blocks)
1958 
1959 def _generator_threegraph(n):
1960     r"""
1961     Given `n` integer, generates the threegraphs of size `n` using nauty
1962     and returns them in a dictionary required for Flag constructors.
1963 
1964     Can also be modified to return hypergraphs for any size, but
1965     larger edge sizes result in too large theories usually.
1966     """
1967     r = 3
1968     for ee in range(binomial(n, r)+1):
1969         for xx in hypergraphs.nauty(ee, n, uniform=r):
1970             yield {'edges': xx}
1971 
1972 def _identify_hypergraph(n, ftype_points, edges):
1973     r"""
1974     Identifies hypergraphs by creating a canonical label for the adjacency
1975     bipartite graph.
1976     """
1977     g = Graph([list(range(n+len(edges))), [(i+n,x) for i,b in enumerate(edges)
      for x in b]],
1978               format='vertices_and_edges')
1979     partt = [[ii] for ii in ftype_points] + \
1980             [[ii for ii in range(n) if ii not in ftype_points]] + \
1981             [list(range(n,n+len(edges)))]
1982     blocks = tuple(g.canonical_label(partition=partt).edges(labels=None, sort=
     True))
1983     ftype_points = tuple(range(len(ftype_points)))
1984     return (n, ftype_points, blocks)
1985 
1986 def _generator_digraph(n):
1987     r"""
1988     Given `n` integer, generates the digraphs of size `n` using nauty
```

```
1989     and returns them in a dictionary required for Flag constructors
1990     """
1991     gen = graphs.nauty_geng(str(n))
1992     for xx in digraphs.nauty_directg(gen):
1993         yield {'edges': tuple(xx.edges(labels=None))}
1994
1995 def _identify_digraph(n, ftype_points, edges):
1996     r"""
1997     Creates a unique identifier for a graph using canonical labelings
1998     """
1999     partition = [[ii] for ii in ftype_points] + [list(set(range(n)).difference
        (set(ftype_points))), ]
2000     g = DiGraph([list(range(n)), edges], format='vertices_and_edges')
2001     blocks = tuple(g.canonical_label(partition=partition).edges(labels=None,
        sort=True))
2002     ftype_points = tuple(range(len(ftype_points)))
2003     return (n, ftype_points, blocks)
2004
2005 def _generator_tournament(n):
2006     r"""
2007     Given `n` integer, generates the tournaments of size `n` using nauty
2008     and returns them in a dictionary required for Flag constructors
2009     """
2010     for xx in digraphs.tournaments_nauty(n):
2011         yield {'edges': tuple(xx.edges(labels=None))}
2012
2013 def _generator_permutation(n):
2014     r"""
2015     Given `n` integer, generates the permutations of objects `n`
2016     and returns the ordering as a binary relation in dictionary
2017     form required for Flag constructors
2018     """
2019     for perm in itertools.permutations(range(n)):
2020         yield {'edges': tuple(itertools.combinations(perm, r=2))}
2021
```

```
2022 def _identify_permutation(n, ftype_points, edges):
2023     r"""
2024     Returns a unique representation of this permutation
2025     """
2026     return (ftype_points, tuple(sorted(edges)))
2027
2028 def _identify_oe_graph(n, ftype_points, edges):
2029     r"""
2030     Identifies ordered edge graphs by creating a canonical label for
2031     the adjacency bipartite graph.
2032     """
2033     g = Graph([list(range(n+len(edges))), [(i+n,x) for i,b in enumerate(edges)
      for x in b]],
2034                 format='vertices_and_edges')
2035     partt = [[ii] for ii in ftype_points] + \
2036             [[ii] for ii in range(n, n+len(edges))] + \
2037             [[ii for ii in range(n) if ii not in ftype_points]]
2038     blocks = tuple(g.canonical_label(partition=partt).edges(labels=None, sort=
    True))
2039     ftype_points = tuple(range(len(ftype_points)))
2040     return (n, ftype_points, blocks)
2041
2042 def _generator_oe_graph(n):
2043     r"""
2044     Given `n` integer, generates the graphs on `n` vertices
2045     with different (non-isomorphic) edge orderings.
2046     """
2047     for xx in graphs.nauty_geng(str(n)):
2048         unordered = tuple(xx.edges(labels=None))
2049         unique = []
2050         for perm in itertools.permutations(unordered):
2051             rel = _identify_oe_graph(n, [], perm)
2052             if rel not in unique:
2053                 unique.append(rel)
2054                 yield {'edges': perm}
```

```
2055
2056 def _generator_ov_graph(n):
2057     r"""
2058     Given `n` integer, generates the graphs on `n` vertices
2059     with different (non-isomorphic) vertex orderings.
2060
2061     This is the same set as the boolean symmetric
2062     nxn matrices with 0s on the diagonal.
2063     """
2064     full = list(itertools.combinations(range(n), int(2)))
2065     for ii in range(binomial(n, 2)+1):
2066         for xx in itertools.combinations(full, int(ii)):
2067             yield {'edges': xx}
2068
2069 def _identify_ov_graph(n, ftype_points, edges):
2070     r"""
2071     Returns a unique representation for this ordered
2072     vertex graph
2073     """
2074     return (n, tuple(ftype_points), tuple(sorted(list(edges))))
2075
2076
2077
2078 def _ramsey_graphs_from_graph(xx, n, excess_edges, two_edge_triples):
2079     r"""
2080     From a graph creates all the Ramsey graphs using nauty multig.
2081
2082     A Ramsey graph here is essentially a multigraph where the edge
2083     colorings are represented by multi edges.
2084     """
2085     from sage.features.nauty import NautyExecutable
2086     import subprocess, select
2087     import shlex
2088     directg_path = NautyExecutable("multig").absolute_filename()
2089     edges = xx.edges(labels=None)
```

```
2090     le = len(edges)
2091     options=" -T -m2 -e{}".format(le+excess_edges)
2092     sub = subprocess.Popen(
2093         shlex.quote(directg_path) + ' {0}'.format(options),
2094         shell=True,
2095         stdout=subprocess.PIPE,
2096         stdin=subprocess.PIPE,
2097         stderr=subprocess.PIPE,
2098         encoding='latin-1'
2099     )
2100     sub.stdin.write(xx.graph6_string())
2101     sub.stdin.close()
2102     unique = []
2103     for ll in sub.stdout:
2104         if ll and ll[0]==str(n):
2105             edges_marked = []
2106             seq = ll[:-1].split(" ")
2107             for ii in range(1, len(seq)//3):
2108                 try:
2109                     ed = (int(seq[ii*3]), int(seq[ii*3 + 1]))
2110                 except:
2111                     print("error on line: \n", ll, "\nhappened at graph: ", xx
    )
2112                     return
2113                 if seq[ii*3 + 2]=="2":
2114                     edges_marked.append(ed)
2115             two_good = True
2116             for trp in two_edge_triples:
2117                 count = sum([(trp[0], trp[1]) in edges_marked,
2118                              (trp[0], trp[2]) in edges_marked,
2119                              (trp[1], trp[2]) in edges_marked])
2120                 if count == 1:
2121                     two_good = False
2122                     break
2123             if two_good:
```

```
2124                 yield {'edges': edges, 'edges_marked': edges_marked}
2125     for line in sub.stderr:
2126         pass
2127     sub.wait()
2128
2129 def _generator_ramsey_graph(n):
2130     r"""
2131     Given `n` integer, generates the graphs on `n` vertices
2132     with some of the edges marked.
2133
2134     This is color-blind, so coloring the complement of the graph
2135     results in the same graph.
2136     """
2137     for xx in graphs.nauty_geng(str(n)):
2138         edges = xx.edges(labels=None)
2139         good = True
2140         two_edge_triples = []
2141         for trp in itertools.combinations(range(n), int(3)):
2142             count = sum([(trp[0], trp[1]) in edges,
2143                          (trp[0], trp[2]) in edges,
2144                          (trp[1], trp[2]) in edges])
2145             if count == 1:
2146                 good = False
2147                 break
2148             if count == 2:
2149                 two_edge_triples.append(trp)
2150         if good:
2151             le = len(edges)
2152             for ii in range(le//2 + 1):
2153                 for rr in _ramsey_graphs_from_graph(xx, n, ii,
    two_edge_triples):
2154                     yield rr
2155
2156 def _identify_ramsey_graph(n, ftype_points, edges, edges_marked):
2157     r"""
```

```
2158    Creates a canonical label for the Ramsey graph and returns
2159    it as the identifier.
2160    """
2161    if len(edges_marked) == len(edges)/2:
2162        edges_marked_alter = [xx for xx in edges if xx not in edges_marked]
2163        g = Graph([list(range(n+len(edges)+len(edges_marked))),
2164                   [(i+n, x) for i,b in enumerate(edges) for x in b] +
2165                  [(i+n+len(edges), x) for i, b in enumerate(edges_marked) for
     x in b]],
2166                  format='vertices_and_edges')
2167        partt = [[ii] for ii in ftype_points] + \
2168                [[ii for ii in range(n) if ii not in ftype_points]] + \
2169                [list(range(n,n+len(edges)))] + \
2170                [list(range(n+len(edges), n+len(edges)+len(edges_marked)))]
2171        blocks = tuple(g.canonical_label(partition=partt).edges(labels=None,
    sort=True))
2172
2173        g_alter = Graph([list(range(n+len(edges)+len(edges_marked_alter))),
2174                   [(i+n, x) for i,b in enumerate(edges) for x in b] +
2175                  [(i+n+len(edges), x) for i, b in enumerate(
    edges_marked_alter) for x in b]],
2176                  format='vertices_and_edges')
2177        partt_alter = [[ii] for ii in ftype_points] + \
2178                [[ii for ii in range(n) if ii not in ftype_points]] + \
2179                [list(range(n,n+len(edges)))] + \
2180                [list(range(n+len(edges), n+len(edges)+len(edges_marked_alter)
    ))]
2181        blocks_alter = tuple(g_alter.canonical_label(partition=partt_alter).
    edges(labels=None, sort=True))
2182        return (n, tuple(range(len(ftype_points))), min(blocks, blocks_alter))
2183    else:
2184        if len(edges_marked) > len(edges)/2:
2185            edges_marked = [xx for xx in edges if xx not in edges_marked]
2186        g = Graph([list(range(n+len(edges)+len(edges_marked))),
2187                   [(i+n, x) for i,b in enumerate(edges) for x in b] +
```

```
2188                  [(i+n+len(edges), x) for i, b in enumerate(edges_marked) for
     x in b]],
2189                  format='vertices_and_edges')
2190        partt = [[ii] for ii in ftype_points] + \
2191                [[ii for ii in range(n) if ii not in ftype_points]] + \
2192                [list(range(n,n+len(edges)))] + \
2193                [list(range(n+len(edges), n+len(edges)+len(edges_marked)))]
2194        blocks = tuple(g.canonical_label(partition=partt).edges(labels=None,
    sort=True))
2195        return (n, tuple(range(len(ftype_points))), blocks)
2196
2197 GraphTheory = CombinatorialTheory('Graph',
2198                                   _generator_graph,
2199                                   _identify_graph,
2200                                   edges=2)
2201
2202 ThreeGraphTheory = CombinatorialTheory('3-Graph',
2203                                        _generator_threegraph,
2204                                        _identify_hypergraph,
2205                                        edges=3)
2206
2207 DiGraphTheory = CombinatorialTheory('DiGraph',
2208                                     _generator_digraph,
2209                                     _identify_digraph,
2210                                     edges=2)
2211
2212 TournamentTheory = CombinatorialTheory('Tournament',
2213                                        _generator_tournament,
2214                                        _identify_digraph,
2215                                        edges=2)
2216 #Note: TournamentTheory is equivalent to
2217 #DiGraphTheory.exclude(DiGraphTheory(2, edges=[[0, 1], [1, 0]]))
2218
2219 PermutationTheory = CombinatorialTheory('Permutation',
2220                                         _generator_permutation,
```

```
2221                                         _identify_permutation,
2222                                         edges=2)
2223
2224 OEGraphTheory = CombinatorialTheory('OEdgeGraph',
2225                                     _generator_oe_graph,
2226                                     _identify_oe_graph,
2227                                     edges=2)
2228
2229 OVGraphTheory = CombinatorialTheory('OEdgeGraph',
2230                                     _generator_ov_graph,
2231                                     _identify_ov_graph,
2232                                     edges=2)
2233
2234 RamseyGraphTheory = CombinatorialTheory('RamseyGraph',
2235                                         _generator_ramsey_graph,
2236                                         _identify_ramsey_graph,
2237                                         edges=2,
2238                                         edges_marked=2)
```

A.2 Flag

```
1 r"""
2 Implementation of Flag, elements of :class:`CombinatorialTheory`
3
4 Cython class for flags and types. Types will be called ftype
5 (short for flag type, to distinguish from the type keyword in python)
6 They are elements of :class:`CombinatorialTheory`. They also behave as
7 basis elements for :class:`FlagAlgebra`, hence basic operations
8 like addition and multiplication are defined.
9
10
11 The class :class:`CombinatorialTheory` acts as a parent for flag elements.
12 Its members are theories, they come with signature, ways to generate
13 elements, ways to check if they are equal. This example uses the
14 GraphTheory object. To find out more about combinatorial theories,
```

```
15 other pre-implemented members, and how to solve problems inside them
16 using flag algebras, see the documentation of
17 :mod:`sage.algebras.flag_algebras`. This file is about flags mainly.
18
19 Formally, for a given theory $T$, and two models $M, N$,
20 with model embedding of $M$ in $N$, $\Theta: M \rightarrow N$, the pair
21 $F = (N, \Theta)$ is a flag, with type $M$. The size of the model is
22 the number of elements in it. Here, the elements of each model form
23 initial segments of the naturals, therefore a model on 3 elements must
24 have elements `range(3)`. The relations in the theory are
25 list of lists from the elements. For example in the theory of graphs,
26 there is only one relational symbol, the edge. This allows to specify
27 a triangle with the edge list `[[0, 1], [0, 2], [1, 2]]`. As combinatorial
28 theories are closed under sub-models, any subset of ${0, 1, 2}$ induces a
29 sub-model. For example, the subset $1$ induces the empty graph with $1$
30 vertex. The embedding of this empty graph into the triangle therefore forms
31 a flag. Notice that the collection of flags $(N, \emptyset)$ is isomorphic
32 to the models of $T$. In this code they are identified, so elements of $T$
33 are simply flags with empty type. Furthermore, to distinguish types from
34 models (and to simplify calculations), the types are identified with flags
35 using the identity embedding $(M, id)$.
36
37
38 The examples will use `GraphTheory`
39
40     sage: from sage.algebras.flag_algebras import GraphTheory
41
42 To create flags from a theory we can call for example ::
43
44     sage: g = GraphTheory(3, edges=[[0, 1]])
45
46 This creates a graph on `3` vertices (equal to `[0, 1, 2]`),
47 and it defines the single edge `[0, 1]`. The result is a Flag
48 `g`. The ftype is a collection of marked vertices (a constant in the theory).
49 For example, we can define the same graph but mark one vertex ::
```

```
50
51     sage: g0 = GraphTheory(3, edges=[[0, 1]], ftype=[0])
52     sage: g1 = GraphTheory(3, edges=[[0, 1]], ftype=[1])
53     sage: g2 = GraphTheory(3, edges=[[0, 1]], ftype=[2])
54
55 Note that marking ‘2‘ defines a different flag than the other two since the
56 only nontrivial automorphism of the graph is the [0, 1] pair transposition ::
57
58     sage: g0==g2
59     False
60     sage: g0==g1
61     True
62
63 On flags we can do various computations. For example we can define an edge ::
64
65     sage: e = GraphTheory(2, edges=[[0, 1]])
66
67 And calculate the sum of the two elements resulting in a
68 FlagAlgebraElement ::
69
70     sage: e+g
71     Flag Algebra Element over Rational Field
72     0   - Flag on 3 points, ftype from [] with edges=[]
73     4/3 - Flag on 3 points, ftype from [] with edges=[[0, 2]]
74     2/3 - Flag on 3 points, ftype from [] with edges=[[0, 2], [1, 2]]
75     1   - Flag on 3 points, ftype from [] with edges=[[0, 1], [0, 2], [1, 2]]
76
77 We can also calculate the product of the elements ::
78
79     sage: e*g
80     Flag Algebra Element over Rational Field
81     1/5  - Flag on 5 points, ftype from [] with edges=[[0, 3], [1, 4]]
82     ...
83     3/10 - Flag on 5 points, ftype from [] with edges=[[0, 2], [0, 3], [0, 4],
        [1, 4], [2, 3], [2, 4], [3, 4]]
```

```
84      1/10 - Flag on 5 points, ftype from [] with edges=[[0, 2], [0, 3], [0, 4],
         [1, 3], [1, 4], [2, 3], [2, 4], [3, 4]]
85
86  In order to add or multiply elements, it is required to have the same ftype
87  and the result will have the same ftype. In the examples above
88  `e, g, e+g, e*g` all have the same empty ftype. It is the default ftype when
89  nothing is provided. `g0, g1, g2` have the same point
90  ftype ::
91
92      sage: g.ftype()
93      Ftype on 0 points with edges=[]
94      sage: g1.ftype()
95      Ftype on 1 points with edges=[]
96
97  This implementation defines ftypes as flags. The only distinction is that
98  an ftype has all it's points part of the ftype, while flags have points
99  outside the ftype.
100
101 To change between ftypes, one can project the flag to a smaller ftype.
102 A projection is defined by an injective map from a smaller ftype into a larger
103 one. For example we can define the following flag with a largeish ftype ::
104
105     sage: gg = GraphTheory(5, edges=[[0, 1], [2, 3]], ftype=[0, 2, 3, 4])
106     sage: gg
107     Flag on 5 points, ftype from [0, 2, 3, 4] with edges=[[0, 1], [2, 3]]
108     sage: gg.ftype()
109     Ftype on 4 points with edges=[[1, 2]]
110
111 The injection is simply a list of the smaller ftype's points inside
112 the larger ftype's points. For example picking `[0, 1, 2]` as the
113 injection, would result in an ftype on `[0, 2, 3]` points with the same
114 edge `[2, 3]` which after renaming to `[0, 1, 2]` would be `[1, 2]`::
115
116     sage: gg.project([0, 1, 2])
117     Flag Algebra Element over Rational Field
```

```
118     1/2 - Flag on 5 points, ftype from [0, 1, 4] with edges=[[0, 3], [1, 4]]
119     sage: gg.project([0, 1, 2]).ftype()
120     Ftype on 3 points with edges=[[1, 2]]
121 
122 The result is normalized, but note that it is isomorphic to the one claimed
123 before. Three points `[0, 1, 4]` and one edge between, `[1, 4]`. If nothing
124 is provided then the flag is projected to the empty ftype ::
125 
126     sage: gg.project()
127     Flag Algebra Element over Rational Field
128     1/15 - Flag on 5 points, ftype from [] with edges=[[0, 3], [1, 4]]
129 
130 For a little extra speed, multiplication and then projection is implemented::
131 
132     sage: gp = GraphTheory(2, ftype=[0])
133     sage: (gp*gp).project()
134     Flag Algebra Element over Rational Field
135     1   - Flag on 3 points, ftype from [] with edges=[]
136     1/3 - Flag on 3 points, ftype from [] with edges=[[0, 2]]
137     0   - Flag on 3 points, ftype from [] with edges=[[0, 2], [1, 2]]
138     0   - Flag on 3 points, ftype from [] with edges=[[0, 1], [0, 2], [1, 2]]
139     sage: gp.mul_project(gp) == (gp*gp).project()
140     True
141 
142 .. SEEALSO::
143     :func:`Flag.__init__`
144     :func:`Flag.ftype`
145     :func:`Flag._add_`
146     :func:`Flag._mul_`
147     :func:`Flag.project`
148     :func:`Flag.mul_project`
149     :class:`CombinatorialTheory`
150     :class:`FlagAlgebra`
151     :class:`FlagAlgebraElement`
152 
```

```
153 AUTHORS:
154
155 - Levente Bodnar (Dec 2023): Initial version
156
157 """
158
159
160 import itertools
161 from sage.rings.rational_field import QQ
162 from cysignals.signals cimport sig_check
163 from sage.structure.element cimport Element
164
165 cpdef _subblock_helper(list points, list block):
166     r"""
167     Helper to find induced substructures
168
169     TESTS::
170
171         sage: _subblock_helped([0, 2], [[0, 1, 2], [2, 4], [1, 1, 2]])
172         [[0, 1, 2]]
173     """
174     cdef bint gd = 0
175     ret = []
176     if len(block)==0:
177         return ret
178     for xx in block:
179         gd = 1
180         for yy in xx:
181             if yy not in points:
182                 gd = 0
183                 break
184         if gd:
185             ret.append([points.index(ii) for ii in xx])
186     return ret
187
```

```
188 cdef class Flag(Element):
189 
190     cdef int _n
191     cdef int _ftype_size
192 
193     cdef list _ftype_points
194     cdef list _not_ftype_points
195     cdef dict _blocks
196     cdef tuple _unique
197 
198     cdef Flag _ftype
199 
200     def __init__(self, theory, n, **params):
201         r"""
202         Initialize a :class:`Flag` element
203 
204         INPUT:
205 
206         - ``theory`` -- :class:`CombinatorialTheory`; the underlying theory,
207             which is also the parent
208         - ``n`` -- integer; the size (number of vertices) of the flag
209         - ``**params`` -- optional parameters; can contain the points of the
210             ftype as a list of points under the name "ftype" of "ftype_points
    ",
211             if not provided then empty ftype is assumed. Can also contain the
212             blocks for each element in the signature, if not provided then an
213             empty block set is assumed.
214 
215         OUTPUT: The resulting flag
216 
217         EXAMPLES::
218 
219         Create a simple GraphTheory triangle ::
220 
221             sage: from sage.algebras.flag_algebras import *
```

```
222             sage: from sage.algebras.flag import Flag
223             sage: Flag(GraphTheory, 3, edges=[[0, 1], [0, 2], [1, 2]])
224             Flag on 3 points, ftype from [] with edges=[[0, 1], [0, 2], [1,
    2]]
225 
226         Create a DiGraphTheory edge out from a pointed ftype ::
227 
228             sage: Flag(DiGraphTheory, 2, edges=[[0, 1]], ftype=[0])
229             Flag on 2 points, ftype from [0] with edges=[[0, 1]]
230 
231         .. NOTE::
232 
233             It is recommended to create flags using the parent's element
    constructor
234 
235         .. SEEALSO::
236 
237             :func:`CombinatorialTheory._element_constructor_`
238         """
239         self._n = int(n)
240 
241         if 'ftype_points' in params:
242             ftype_points = params['ftype_points']
243         elif 'ftype' in params:
244             ftype_points = params['ftype']
245         else:
246             ftype_points = []
247 
248         self._ftype_size = len(ftype_points)
249         self._ftype_points = list(ftype_points)
250         self._not_ftype_points = None
251         self._blocks = {}
252         for xx in theory._signature.keys():
253             if xx in params:
254                 self._blocks[xx] = [list(yy) for yy in params[xx]]
```

```
255             else:
256                 self._blocks[xx] = []
257         self._unique = None
258 
259         self._ftype = None
260         Element.__init__(self, theory)
261 
262     def _repr_(self):
263         r"""
264         Return a nice representation.
265 
266         If it is an ftype (all the points are part of the ftype), then
267         the text changes to indicate this fact.
268 
269         EXAMPLES::
270 
271         For flags ::
272 
273             sage: from sage.algebras.flag_algebras import *
274             sage: GraphTheory(3)
275             Flag on 3 points, ftype from [] with edges=[]
276 
277         For ftypes ::
278 
279             sage: ThreeGraphTheory(4, ftype_points=[0, 1, 2, 3], edges=[[0, 1,
     2]])
280             Ftype on 4 points with edges=[[0, 1, 2]]
281         """
282         blocks = self.blocks()
283         strblocks = ', '.join([xx+'='+str(blocks[xx]) for xx in blocks.keys()
    ])
284         if self.is_ftype():
285             return 'Ftype on {} points with {}'.format(self.size(), strblocks)
286         return 'Flag on {} points, ftype from {} with {}'.format(self.size(),
    self.ftype_points(), strblocks)
```

```
287
288     def raw_numbers(self):
289         r"""
290         Return a list of numbers uniquely describing this flag.
291
292         This is used in saving and loading calculations
293
294         EXAMPLES::
295
296             sage: from sage.algebras.flag_algebras import *
297             sage: GraphTheory(2, edges=[[0, 1]], ftype=[0]).raw_numbers()
298             [2, 0, 15, 0, 1, 15]
299
300         """
301         numbers = [self.size()] + self.ftype_points() + [15]
302         blocks = self.blocks()
303         for xx in blocks:
304             for yy in blocks[xx]:
305                 numbers += yy
306             numbers.append(15)
307         return numbers
308
309     cpdef subflag(self, points=None, ftype_points=None):
310         r"""
311         Returns the induced subflag.
312
313         The resulting sublaf contains the union of points and ftype_points
314         and has ftype constructed from ftype_points.
315
316         INPUT:
317
318         - ``points`` -- list (default: `None`); the points inducing the
    subflag.
319             If not provided (or `None`) then this is the entire vertex set, so
320             only the ftype changes
```

```
321         - ``ftype_points`` list (default: `None`); the points inducing the
    ftype
322             of the subflag. If not provided (or `None`) then the original
    ftype
323             point set is used, so the result of the ftype will be the same
324 
325         OUTPUT: The induced sub Flag
326 
327         EXAMPLES::
328 
329         Same ftype ::
330 
331             sage: from sage.algebras.flag_algebras import *
332             sage: g = GraphTheory(3, edges=[[0, 1]], ftype=[0])
333             sage: g.subflag([0, 2])
334             Flag on 2 points, ftype from [0] with edges=[]
335 
336         Only change ftype ::
337 
338             sage: g.subflag(ftype_points=[0, 1])
339             Flag on 3 points, ftype from [0, 1] with edges=[[0, 1]]
340 
341         .. NOTE::
342 
343             As the ftype points can be chosen, the result can have different
344             ftype as self.
345 
346         TESTS::
347 
348             sage: g.subflag()==g
349             True
350         """
351         if ftype_points==None:
352             ftype_points = self._ftype_points
353         if points==None:
```

```
354                 points = list(range(self._n))
355             else:
356                 points = [ii for ii in range(self._n) if (ii in points or ii in
        ftype_points)]
357             if len(points)==self._n and ftype_points==self._ftype_points:
358                 return self
359             blocks = {xx: _subblock_helper(points, self._blocks[xx]) for xx in
        self._blocks.keys()}
360             new_ftype_points = [points.index(ii) for ii in ftype_points]
361             return self.__class__(self.parent(), len(points), ftype=
        new_ftype_points, **blocks)
362 
363         def combinatorial_theory(self):
364             r"""
365             Returns the combinatorial theory this flag is a member of
366 
367             This is the same as the parent.
368 
369             .. SEEALSO::
370 
371                 :func:`theory`
372                 :func:`parent`
373             """
374             return self.parent()
375 
376         theory = combinatorial_theory
377 
378         def as_flag_algebra_element(self, basis=QQ):
379             r"""
380             Transforms this `Flag` to a `FlagAlgebraElement` over a given basis
381 
382             INPUT:
383 
384             - ``basis`` -- Ring (default: `QQ`); the base of
385                 the FlagAlgebra where the target will live.
```

```
386
387        OUTPUT: A ‘FlagAlgebraElement‘ representing this ‘Flag‘
388
389        .. SEEALSO::
390
391            :class:‘FlagAlgebra‘
392            :func:‘FlagAlgebra._element_constructor_‘
393            :class:‘FlagAlgebraElement‘
394        """
395        from sage.algebras.flag_algebras import FlagAlgebra
396        targ_alg = FlagAlgebra(basis, self.theory(), self.ftype())
397        return targ_alg(self)
398
399    afae = as_flag_algebra_element
400
401    def as_operand(self):
402        r"""
403        Turns this ‘Flag‘ into a ‘FlagAlgebraElement‘ so operations can be
    performed on it
404
405        .. SEEALSO::
406
407            :func:‘as_flag_algebra_element‘
408        """
409        return self.afae(QQ)
410
411    cpdef size(self):
412        r"""
413        Returns the size of the vertex set of this Flag.
414
415        OUTPUT: integer, the number of vertices.
416
417        EXAMPLES::
418
419        This is the size parameter in the ‘Flag‘ initialization ::
```

```
420
421             sage: from sage.algebras.flag_algebras import *
422             sage: GraphTheory(4).size()
423             4
424         """
425         return self._n
426
427     vertex_number = size
428
429     cpdef blocks(self, as_tuple=False):
430         r"""
431         Returns the blocks
432
433         INPUT:
434
435         - ``as_tuple`` -- boolean (default: `False`); if the result should
436             contain the blocks as a tuple
437
438         OUTPUT: A dictionary, one entry for each element in the signature
439             and list (or tuple) of the blocks for that signature.
440         """
441         if as_tuple:
442             ret = {}
443             for xx in self._blocks:
444                 ret[xx] = tuple([tuple(yy) for yy in self._blocks[xx]])
445             return ret
446         return self._blocks
447
448     cpdef ftype(self):
449         r"""
450         Returns the ftype of this `Flag`
451
452         EXAMPLES::
453
454         Ftype of a pointed triangle is just a point ::
```

```
455
456             sage: from sage.algebras.flag_algebras import *
457             sage: pointed_triangle = GraphTheory(3, edges=[[0, 1], [0, 2], [1,
     2]], ftype=[0])
458             sage: pointed_triangle.ftype()
459             Ftype on 1 points with edges=[]
460
461         And with two points it is ::
462
463             sage: two_pointed_triangle = GraphTheory(3, edges=[[0, 1], [0, 2],
     [1, 2]], ftype=[0, 1])
464             sage: two_pointed_triangle.ftype()
465             Ftype on 2 points with edges=[[0, 1]]
466
467         .. NOTE::
468
469             This is essentially the subflag, but the order of points matter.
    The result is saved
470             for speed.
471
472         .. SEEALSO::
473
474             :func:`subflag`
475         """
476         if self._ftype==None:
477             if self.is_ftype():
478                 self._ftype = self
479             self._ftype = self.subflag([])
480         return self._ftype
481
482     cpdef ftype_points(self):
483         r"""
484         The points of the ftype inside self.
485
486         This gives an injection of ftype into self
```

```
487
488         OUTPUT: list of integers
489
490         EXAMPLES::
491
492             sage: from sage.algebras.flag_algebras import *
493             sage: two_pointed_triangle = GraphTheory(3, edges=[[0, 1], [0, 2],
    [1, 2]], ftype=[0, 1])
494             sage: two_pointed_triangle.ftype_points()
495             [0, 1]
496
497         .. SEEALSO::
498
499             :func:`__init__`
500         """
501         return self._ftype_points
502
503     cpdef not_ftype_points(self):
504         r"""
505         This is a helper function, caches the points that are not
506         part of the ftype.
507         """
508         if self._not_ftype_points != None:
509             return self._not_ftype_points
510         self._not_ftype_points = [ii for ii in range(self.size()) if ii not in
    self._ftype_points]
511         return self._not_ftype_points
512
513     def unique(self):
514         r"""
515         This returns a unique identifier that can equate isomorphic
516         objects
517
518         EXAMPLES::
519
```

```
520         Isomorphic graphs have the same :func:`unique` value ::
521
522             sage: from sage.algebras.flag_algebras import *
523             sage: b1 = [[0, 1], [0, 2], [0, 4], [1, 3], [2, 4]]
524             sage: b2 = [[0, 4], [1, 2], [1, 3], [2, 3], [3, 4]]
525             sage: g1 = GraphTheory(5, edges=b1)
526             sage: g2 = GraphTheory(5, edges=b2)
527             sage: g1.unique() == g2.unique()
528             True
529
530         .. NOTE::
531
532             The value returned here depends on the values of
533             the parent `CombinatorialTheory`
534
535         .. SEEALSO::
536
537             :func:`theory`
538             :func:`CombinatorialTheory.identify`
539             :func:`__eq__`
540         """
541         if self._unique==None:
542             self._unique = self.theory().identify(
543                 self._n, self._ftype_points, **self._blocks)
544         return self._unique
545
546     cpdef is_ftype(self):
547         r"""
548         Returns `True` if this flag is an ftype.
549
550         .. SEEALSO::
551
552             :func:`_repr_`
553         """
554         return self._n == self._ftype_size
```

```
555
556     def _add_(self, other):
557         r"""
558         Add two Flags together
559
560         The flags must have the same ftype. Different sizes are
561             all shifted to the larger one.
562
563         OUTPUT: The :class:`FlagAlgebraElement` object,
564             which is the sum of the two parameters
565
566         EXAMPLES::
567
568         Adding to self is 2*self ::
569
570             sage: from sage.algebras.flag_algebras import *
571             sage: g = GraphTheory(3)
572             sage: g+g==2*g
573             True
574
575         Adding two distinct elements with the same size gives a vector
576         with exactly two `1` entries ::
577
578             sage: h = GraphTheory(3, edges=[[0, 1]])
579             sage: (g+h).values()
580             (1, 1, 0, 0)
581
582         Adding with different size the smaller flag
583         is shifted to have the same size ::
584
585             sage: e = GraphTheory(2)
586             sage: (e+h).values()
587             (1, 5/3, 1/3, 0)
588
589         .. SEEALSO::
```

```
590
591             :func:`FlagAlgebraElement._add_`
592             :func:`__lshift__`
593
594         """
595         if self.ftype()!=other.ftype():
596             raise TypeError("The terms must have the same ftype")
597         return self.afae()._add_(other.afae())
598
599     def _sub_(self, other):
600         r"""
601         Subtract a Flag from `self`
602
603         The flags must have the same ftype. Different sizes are
604             all shifted to the larger one.
605
606         EXAMPLES::
607
608             sage: from sage.algebras.flag_algebras import *
609             sage: g = GraphTheory(2)
610             sage: h = GraphTheory(3, edges=[[0, 1]])
611             sage: (g-h).values()
612             (1, -1/3, 1/3, 0)
613
614         .. SEEALSO::
615
616             :func:`_add_`
617             :func:`__lshift__`
618             :func:`FlagAlgebraElement._sub_`
619
620         """
621         if self.ftype()!=other.ftype():
622             raise TypeError("The terms must have the same ftype")
623         return self.afae()._sub_(other.afae())
624
```

```
625     def _mul_(self, other):
626         r"""
627         Multiply two flags together.
628 
629         The flags must have the same ftype. The result
630         will have the same ftype and size
631         `self.size() + other.size() - self.ftype().size()`
632 
633         OUTPUT: The :class:`FlagAlgebraElement` object,
634             which is the product of the two parameters
635 
636         EXAMPLES::
637 
638         Pointed edge multiplied by itself ::
639 
640             sage: from sage.algebras.flag_algebras import *
641             sage: pe = GraphTheory(2, edges=[[0, 1]], ftype=[0])
642             sage: (pe*pe).values()
643             (0, 0, 0, 0, 1, 1)
644 
645         .. SEEALSO::
646 
647             :func:`FlagAlgebraElement._mul_`
648             :func:`mul_project`
649             :func:`CombinatorialTheory.mul_project_table`
650 
651         TESTS::
652 
653             sage: sum((pe*pe*pe*pe).values())
654             11
655             sage: e = GraphTheory(2)
656             sage: (e*e).values()
657             (1, 2/3, 1/3, 0, 2/3, 1/3, 0, 0, 1/3, 0, 0)
658         """
659         if self.ftype()!=other.ftype():
```

```
660             raise TypeError("The terms must have the same ftype")
661         return self.afae()._mul_(other.afae())
662 
663     def __lshift__(self, amount):
664         r"""
665         `FlagAlgebraElement`, equal to this, with size is shifted by the
    amount
666 
667         EXAMPLES::
668 
669         Edge shifted to size `3` ::
670 
671             sage: from sage.algebras.flag_algebras import *
672             sage: edge = GraphTheory(2, edges=[[0, 1]])
673             sage: (edge<<1).values()
674             (0, 1/3, 2/3, 1)
675 
676         .. SEEALSO::
677 
678             :func:`FlagAlgebraElement.__lshift__`
679         """
680         return self.afae().__lshift__(amount)
681 
682     def __truediv__(self, other):
683         r"""
684         Divide by a scalar
685 
686         INPUT:
687 
688         - ``other`` -- number; any number such that `1` can be divided with
    that
689 
690         OUTPUT: The `FlagAlgebraElement` resulting from the division
691 
692         EXAMPLES::
```

```
693
694         Divide by `2` ::
695
696             sage: from sage.algebras.flag_algebras import *
697             sage: g = GraphTheory(3)
698             sage: (g/2).values()
699             (1/2, 0, 0, 0)
700
701         Even for `x` symbolic `1/x` is defined, so the division is understood
    ::
702             sage: var('x')
703             x
704             sage: g = GraphTheory(2)
705             sage: g/x
706             Flag Algebra Element over Symbolic Ring
707             1/x - Flag on 2 points, ftype from [] with edges=[]
708             0   - Flag on 2 points, ftype from [] with edges=[[0, 1]]
709
710         .. NOTE::
711
712             Dividing by `Flag` or `FlagAlgebraElement` is not allowed, only
713             numbers such that the division is defined in some extension
714             of the rationals.
715
716         .. SEEALSO::
717
718             :func:`FlagAlgebraElement.__truediv__`
719         """
720         return self.afae().__truediv__(other)
721
722     def __eq__(self, other):
723         r"""
724         Compare two flags for == (equality)
725
726         .. SEEALSO::
```

```
727
728             :func:`unique`
729             :func:`theory`
730             :func:`CombinatorialTheory.identify`
731         """
732         if type(other)!=type(self):
733             return False
734         if self.parent()!=other.parent():
735             return False
736         return self.unique() == other.unique()
737 
738     def __lt__(self, other):
739         r"""
740         Compare two flags for < (proper induced inclusion)
741 
742         Returns true if self appears as a proper induced structure
743         inside other.
744 
745         .. SEEALSO::
746 
747             :func:`__le__`
748         """
749         if type(other)!=type(self):
750             return False
751         if self.parent()!=other.parent():
752             return False
753         if self.size()>=other.size():
754             return False
755         if self.ftype() != other.ftype():
756             return False
757         for subp in itertools.combinations(other.not_ftype_points(), self.size
    ()-self.ftype().size()):
758             if other.subflag(subp).__eq__(self):
759                 return True
760         return False
```

```
761
762     def __le__(self, other):
763         r"""
764         Compare two flags for <= (induced inclusion)
765
766         Returns true if self appears as an induced structure inside
767         other.
768
769         EXAMPLES::
770
771         Edge appears in a 4 star ::
772
773             sage: from sage.algebras.flag_algebras import *
774             sage: star = GraphTheory(4, edges=[[0, 1], [0, 2], [0, 3]])
775             sage: edge = GraphTheory(2, edges=[[0, 1]])
776             sage: edge <= star
777             True
778
779         The ftypes must agree ::
780
781             sage: p_edge = GraphTheory(2, edges=[[0, 1]], ftype_points=[0])
782             sage: p_edge <= star
783             False
784
785         But when ftypes agree, the inclusion must respect it ::
786
787             sage: pstar = star.subflag(ftype_points=[0])
788             sage: sub1 = GraphTheory(3, ftype=[0], edges=[[0, 1], [0, 2]])
789             sage: sub1 <= pstar
790             True
791             sage: sub2 = GraphTheory(3, ftype=[1], edges=[[0, 1], [0, 2]])
792             sage: sub2 <= pstar
793             False
794
795         .. SEEALSO::
```

```
796
797             :func:`__lt__`
798             :func:`__eq__`
799             :func:`unique`
800         """
801         return self==other or self<other
802
803     def __hash__(self):
804         r"""
805         A hash based on the unique identifier
806         so this is compatible with `__eq__`.
807         """
808         return hash(self.unique())
809
810     def __getstate__(self):
811         r"""
812         Saves this flag to a dictionary
813         """
814         dd = {'theory': self.theory(),
815               'n': self._n,
816               'ftype_points': self._ftype_points,
817               'blocks':self._blocks,
818               'unique':self._unique}
819         return dd
820
821     def __setstate__(self, dd):
822         r"""
823         Loads this flag from a dictionary
824         """
825         self._set_parent(dd['theory'])
826         self._n = dd['n']
827         self._ftype_points = dd['ftype_points']
828         self._ftype_size = len(self._ftype_points)
829         self._not_ftype_points = None
830         self._blocks = dd['blocks']
```

```
831         self._unique = dd['unique']
832 
833     def project(self, ftype_inj=tuple()):
834         r"""
835         Project this `Flag` to a smaller ftype
836 
837 
838         INPUT:
839 
840         - ``ftype_inj`` -- tuple (default: (, )); the injection of the
841             projected ftype inside the larger ftype
842 
843         OUTPUT: the `FlagAlgebraElement` resulting from the projection
844 
845         EXAMPLES::
846 
847         If the center of a cherry is flagged, then the projection has
848         coefficient 1/3 ::
849 
850             sage: from sage.algebras.flag_algebras import *
851             sage: p_cherry = GraphTheory(3, edges=[[0, 1], [0, 2]],
    ftype_points=[0])
852             sage: p_cherry.project().values()
853             (0, 0, 1/3, 0)
854 
855         .. NOTE::
856 
857             If `ftype_inj==tuple(range(self.ftype().size()))` then this
858             does nothing.
859 
860         .. SEEALSO::
861 
862             :func:`FlagAlgebraElement.project`
863         """
864         return self.afae().project(ftype_inj)
```

```
865
866     def mul_project(self, other, ftype_inj=tuple()):
867         r"""
868         Multiply self with other, and the project the result.
869
870         INPUT:
871
872         - ``ftype_inj`` -- tuple (default: (, )); the injection of the
873             projected ftype inside the larger ftype
874
875         OUTPUT: the `FlagAlgebraElement` resulting from the multiplication
876             and projection
877
878         EXAMPLES::
879
880         Pointed edge multiplied with itself and projected ::
881
882             sage: from sage.algebras.flag_algebras import *
883             sage: p_edge = GraphTheory(2, edges=[[0, 1]], ftype_points=[0])
884             sage: p_edge.mul_project(p_edge).values()
885             (0, 0, 1/3, 1)
886
887         .. NOTE::
888
889             If `ftype_inj==tuple(range(self.ftype().size()))` then this
890             is the same as usual multiplication.
891
892         .. SEEALSO::
893
894             :func:`_mul_`
895             :func:`project`
896             :func:`FlagAlgebraElement.mul_project`
897         """
898         return self.afae().mul_project(other, ftype_inj)
899
```

```
900     def density(self, other):
901         r"""
902         The density of self in other.
903 
904         Randomly choosing self.size() points in other, the
905         probability of getting self.
906 
907         EXAMPLES::
908 
909         Density of an edge in the cherry graph is 2/3 ::
910 
911             sage: from sage.algebras.flag_algebras import *
912             sage: cherry = GraphTheory(3, edges=[[0, 1], [0, 2]])
913             sage: edge = GraphTheory(2, edges=[[0, 1]])
914             sage: cherry.density(edge)
915             2/3
916 
917         .. SEEALSO::
918 
919             :func:`FlagAlgebraElement.density`
920         """
921         safae = self.afae()
922         oafae = safae.parent(other)
923         return self.afae().density(other)
924 
925     def _ftypes_inside(self, target):
926         r"""
927         Returns the possible ways self ftype appears in target
928 
929         INPUT:
930 
931         - ``target`` -- Flag; the flag where we are looking for copies of self
932 
933         OUTPUT: list of Flags with ftype matching as self, not necessarily
    unique
```

```
934         """
935         ret = []
936         lrp = list(range(target.size()))
937         for ftype_points in itertools.permutations(range(target.size()), self.
    _n):
938             if target.subflag(ftype_points, ftype_points)==self:
939                 ret.append(target.subflag(lrp, ftype_points))
940         return ret
941 
942     cpdef densities(self, n1, n1flgs, n2, n2flgs, ftype_remap, large_ftype,
    small_ftype):
943         r"""
944         Returns the density matrix, indexed by the entries of `n1flgs` and `
    n2flgs`
945 
946         The matrix returned has entry `(i, j)` corresponding to the
    possibilities of
947         `n1flgs[i]` and `n2flgs[j]` inside self, projected to the small ftype.
948 
949         This is the same as counting the ways we can choose `n1` and `n2`
    points
950         inside `self`, such that the two sets cover the entire `self.size()`
    point
951         set, and calculating the probability that the overlap induces an ftype
952         isomorphic to `large_ftype` and the points sets are isomorphic to
953         `n1flgs[i]` and `n2flgs[j]`.
954 
955         INPUT:
956 
957         - ``n1`` -- integer; the size of the first flag list
958         - ``n1flgs`` -- list of flags; the first flag list (each of size `n1`)
959         - ``n2`` -- integer; the size of the second flag list
960         - ``n2flgs`` -- list of flags; the second flag list (each of size `n2
    `)
961         - ``ftype_remap`` -- list; shows how to remap `small_ftype` into `
```

```
    large_ftype`
962         - ``large_ftype`` -- ftype; the ftype of the overlap
963         - ``small_ftype`` -- ftype; the ftype of self
964 
965         OUTPUT: a sparse matrix corresponding with the counts
966 
967         .. SEEALSO::
968 
969             :func:`CombinatorialTheory.mul_project_table`
970             :func:`FlagAlgebra.mul_project_table`
971             :func:`FlagAlgebraElement.mul_project`
972         """
973         cdef int N = self._n
974         cdef int small_size = small_ftype.size()
975         cdef int large_size = large_ftype.size()
976         cdef int ctr = 0
977         cdef bint chk = 0
978 
979         ret = {}
980         small_points = self._ftype_points
981         for difference in itertools.permutations(self.not_ftype_points(),
    large_size - small_size):
982             sig_check()
983             large_points = [0]*len(ftype_remap)
984             for ii in range(len(ftype_remap)):
985                 vii = ftype_remap[ii]
986                 if vii<small_size:
987                     large_points[ii] = small_points[vii]
988                 else:
989                     large_points[ii] = difference[vii-small_size]
990             ind_large_ftype = self.subflag([], ftype_points=large_points)
991             if ind_large_ftype==large_ftype:
992                 not_large_points = [ii for ii in range(N) if ii not in
    large_points]
993                 for n1_extra_points in itertools.combinations(not_large_points
```

```
, n1 - large_size):
994                     n2_extra_points = [ii for ii in not_large_points if ii not
 in n1_extra_points]
995                     try:
996                         n1_ind = n1flgs.index(self.subflag(n1_extra_points,
ftype_points=large_points))
997                     except ValueError:
998                         subf = self.subflag(n1_extra_points, ftype_points=
large_points)
999                         raise ValueError("Could not find \n", subf, "\nin the
list of ", \
1000                                          n1, " sized flags with ", large_ftype
, \
1001                                          ".\nThis can happen if the generator
and identifier ",\
1002                                          "(from the current
CombinatorialTheory) is incompatible, ",\
1003                                          "or if the theory is not heredetary")
1004                    try:
1005                        n2_ind = n2flgs.index(self.subflag(n2_extra_points,
ftype_points=large_points))
1006                    except:
1007                        subf = self.subflag(n2_extra_points, ftype_points=
large_points)
1008                        raise ValueError("Could not find \n", subf, "\nin the
list of ", \
1009                                          n2, " sized flags with ", large_ftype
, \
1010                                          ".\nThis can happen if the generator
and identifier ",\
1011                                          "(from the current
CombinatorialTheory) is incompatible, ",\
1012                                          "or if the theory is not heredetary")
1013                    try:
1014                        ret[(n1_ind, n2_ind)] += 1
```

```
1015                     except:
1016                         ret[(n1_ind, n2_ind)] = 1
1017         return (len(n1flgs), len(n2flgs), ret)
```