KU LEUVEN

FACULTEIT INGENIEURSWETENSCHAPPEN

A path to compute the 9th Dedekind Number using FPGA Supercomputing

And the first computation of all intervals in the boolean lattice of 7 dimensions

Lennart Van Hirtum

Thesis voorgedragen tot het behalen van de graad van Master of Science in de ingenieurswetenschappen: computerwetenschappen, hoofdoptie Artificiële intelligentie

Promotor:
Prof. dr. Patrick De Causmaecker

Assessoren:
Prof. dr. Ir. Tias Guns
dr. Samuel Kolb

Begeleider:
Jens Goemaere

Academiejaar 2020 – 2021

Preface

I would like to thank my promoter Patrick and my mentor Jens. Their continual involvement and expertise were invaluable for achieving the results in this thesis. I would also like to thank the jury for reading the text. I'm incredibly grateful for having been able to work on this topic. It has truly been my passion for the last year, and it's hopefully an excellent beginning of my academic career.
I'd also like to thank my parents, my sister, my partner, and my friends for supporting me and proofreading this thesis.

Lennart Van Hirtum

Contents

Preface i

Abstract iv

List of Figures and Tables vi

List of Abbreviations and Symbols viii

1 Terminology 1

2 Introduction 3

2.1 C++ Code . . . 4

3 Literature Background 7

3.1 Boolean Functions . . . 7

3.2 Monotonic Boolean Functions . . . 8

3.3 AntiChains . . . 9

3.4 Monotonic - AntiChain duality . . . 9

3.5 Partial ordering on MBFs . . . 9

3.6 Joins, Meets, and the Distributive Lattice of MBFs . . . 10

3.7 Intervals . . . 10

4 Programming Background 13

4.1 SIMD Instructions . . . 14

5 Overview of Operations 15

5.1 Boolean Function Representation . . . 15

5.2 Operations applicable to any Boolean Function . . . 16

5.3 Efficient operations for MBFs . . . 23

5.4 MBF - AntiChain Conversion . . . 26

5.5 Intervals . . . 26

5.6 Interval Iteration . . . 27

6 Finding all Equivalence Classes for 7 Variables 31

6.1 Parallelization and custom `set` Data Structure . . . 32

7 Precomputed values Data Structure 37

7.1 Considerations for large data structure for 7 Variables . . . 38

7.2 Iterating child equivalence classes . . . 39

8 Intervals of D(7) 41

8.1 Naive interval size computation from $\perp$ . . . 41

8.2 Incremental interval size computation 41
8.3 AVX and AVX-512 optimisations 48
8.4 Interval Sizes Performance 48
8.5 Verifying the interval sizes by computing $D(8)$ 49

9 Computing D(9) using P-coefficients **51**
9.1 Work estimate 53
9.2 Computing the P-coefficients 53
9.3 CountConnected 53

10 FPGA Acceleration **63**

11 Discussion and Conclusion **67**

12 Future Work **69**

13 Acknowledgements **71**

Bibliography **73**

Abstract

English: In this thesis we present four important and novel steps towards computing the Ninth Dedekind Number, resulting in the possibility of computing it in a feasible amount of time given the proper hardware. For each of these steps we've been able to outperform all existing methods by orders of magnitude due to a strong focus on efficient operations, efficient data structures and low-level optimizations.

First of all, we present a new method for finding all Monotonic Boolean Function Equivalence Classes in 7 Variables. Using this method we can generate all 490'013'148 Equivalence Classes (and thereby recomputing R(7)) in just 10 minutes on a 4-core CPU, compared to nearly 7 years of CPU time spread over several computers for previous methods. Thereby making it possible to work with 7-variable MBFs on smaller home computers, instead of it being relegated to large supercomputers.

Secondly, we provide a new method for computing the interval size $|[\bot, \alpha]|$ incrementally for every Equivalence Class with 7 variables, a feat never achieved before. We've been able to do this on a single 36-core server from the Genius SuperCluster in 6.5 hours. Using these intervals we've also been able to recompute D(8) in an alternative way. While there were attempts by others, they haven't finished at the time of writing. The method described in this thesis is actually a bit more generic than that, allowing incremental intervals to be computed not only starting from $\bot$, but from any lower bound.

Thirdly, using a cache-friendly data structure and by eliminating expensive operations, we've been able to get the time for the P-Coëfficient method for computing the 8th Dedekind Number down to just 6 minutes on a 4-core machine, compared to most recent attempts which range from 8-12 hours do do the same. This shows the importance of properly optimizing the code, and choosing a language that allows the programmer to do this.

Finally, we propose a way for moving part of the P-Coëfficient computation to a dedicated block of hardware synthesized on High Performance Computing FPGAs, to dramatically speed up this most expensive operation. This finally brings down the total computing time from years on one of the largest supercomputers in Belgium, to months on a modestly sized FPGA cluster. In our final estimate, we believe it to be possible to compute D(9) in 2.5 months, using a cluster of 32 modern High-Performance FPGA cards. Thereby allowing us to make the bold promise of having computed D(9) by the end of 2021.

Nederlands: In deze thesis presenteren we vier belangrijke en nieuwe stappen richting de berekening van het 9e Dedekindgetal. Hierdoor maken we het mogelijk om het 9e Dedekindgetal te berekenen op afzienbare tijd indien we de nodige hardware ter beschikking krijgen. Voor elk van deze stappen zijn we er in geslaagd de bestaande methoden voorbij te streven in ordegroottes van performantie. Dit hebben we kunnen bereiken door een sterke focus te leggen op efficiente operatoren, efficiënte datastructuren en low-level optimalisaties.

Om te beginnen presenteren we een nieuwe methode voor het vinden van alle Equivalentieklassen van Monotone Booleaanse Functies in 7 variabelen. Met deze methode kunnen we de 490'013'148 Equivalentieklassen (en daarbij een herberekening van R(7)) vinden in slechts 10 minuten. Vergelijk dat met de vorige bestaande methode, die er bijna 7 jaar aan CPU-tijd over heeft gedaan, verspreid over verschillende computers. Hierdoor hebben we het mogelijk gemaakt om efficient te werken met Monotone Booleaanse Functies in 7 variabelen op gebruikelijke computers thuis, in plaats van te moeten grijpen naar grote supercomputers.

De tweede stap is een nieuwe methode voor het incrementeel berekenen van de intervalgroottes $|[\bot, \alpha]|$ voor elke Equivalentieklasse, iets wat nog nooit tevoren gelukt was. We zijn erin geslaagd dit te doen op één enkele 36-core server van de Genius SuperCluster in slechts 6.5 uur. Met deze intervalgroottes hebben we ook D(8) kunnen herberekenen via een alternatieve weg. Er waren pogingen door anderen, maar zover we weten is nog niemand er in geslaagd. De methode uit dit deel is zelfs net iets generieker, en staat toe om niet alleen intervalgroottes te berekenen startend van $\bot$, maar van eender welke ondergrens te kunnen vertrekken.

Ten derde, door het gebruik van een cache-friendly datastructuur, en het elimineren van dure operaties, is het gelukt om de tijd voor de berekening van D(8) met de P-Coëfficienten methode naar slechts 6 minuten te brengen. Dit is een grote verbetering vergeleken met de bestaande pogingen, die er meestal 8-12 uur over doen. Dit toont het belang van goede optimalisatie van de code, en het kiezen van een programmeertaal die de programmeur toestaat dit soort optimalisaties te doen.

Ten slotte stellen we een manier voor om een deel van de berekening van de P-Coëfficienten-methode te verplaatsen naar een gespecialiseerd stuk hardware, in de vorm van High Performance FPGAs. Dit helpt ons om de zwaarste stap aanzienlijk te versnellen en maakt het eindelijk mogelijk om D(9) te berekenen binnen enkele maanden op een middelgrote FPGA cluster, in plaats van jaren op de grootste supercomputers van België. Onze recentste schatting voor de berekening van D(9) geeft 2.5 maanden, door gebruik te maken van een cluster van 32 moderne High-Performance FPGA kaarten. We kunnen dus beloven het 9e dedekind getal berekend te hebben tegen het einde van 2021.

List of Figures and Tables

List of Figures

3.1 MBF Example 8
3.2 MBF - AntiChain Duality 10
3.3 Full Lattice Example 11
3.4 Interval Example 12

5.1 varMove Example 17
5.2 varSwap Example 18
5.3 monotonizeDown Example 20
5.4 Powerset iteration example 21
5.5 Boolean Function Hypergraph representation 22
5.6 Canonize Vertex Distinguishing 23
5.7 Predecessor 24
5.8 Successor 25
5.9 Dual MBF 25
5.10 MBF Iteration Structure 30

6.1 Example Equivalence Classes for 4 Variables 34
6.2 BufferedSet vs BakedSet 36

7.1 Structure of one map element. Bit widths shown. 38
7.2 downLink Structure 39

8.1 Interval Example 42
8.2 Smaller interval with known size 43
8.3 Nodes blocked by adding BCE 44
8.4 Leftover nodes to count choices for 45
8.5 Subsets of Nonconflicting elements can be counted with powers of 2 . . 46
8.6 Widest AntiChain, to maximize powers of 2 47
8.7 IntervalSize Performance Measurements 49

9.1 Child Class Counts 54
9.2 CountConnected Example 55
9.3 CountConnected Difficulty 58

9.4 Connection Count Distribution 60
9.5 Singleton Count Distribution 60
9.6 Runtime Distribution in Cycles 61

10.1 CountConnected Core 65
10.2 PCoëff Core 65
10.3 Example naive parallel PCoëff processing. 66

List of Tables

2.1 Known Dedekind Numbers 3
2.2 Known Equivalence Class Counts 3

3.1 Boolean Function Example 8

4.1 x86 SIMD instruction sets by Intel 14

5.1 Example Boolean Function Bitset Representation 15
5.2 Canonization Benchmarks 23

9.1 P-Coëff efficiency 53
9.2 CountConnected Method Timings 59
9.3 Singleton and Iteration Statistics 61

List of Abbreviations and Symbols

Abbreviations

BF	Boolean Function
MBF	Monotonic Boolean Function
AC	AntiChain
SIMD	Single-Instruction Stream Multiple Data Stream
FPGA	Field-Programmable Gate Array
GPGPU	General Purpose Graphical Processing Unit
HPC	High Performance Computing
VSC	Vlaams SuperComputing Centrum (Flemish SuperComputing Center)

Symbols

N	Refers to the number of Variables of the Boolean Function
A_n	Officially the set of all AntiChains, though we use it for the set of all MBFs too depending on the context.
R_n	The set of all Equivalence Class Representatives
D(N)	The N-th dedekind number. D(N) = $\lvert A_n \rvert$
R(N)	The N-th equivalence class count number. R(N) = $\lvert R_n \rvert$
α, β, γ	MBFs or AntiChains
a, b,...	Variables, in bitset representation correspond to numeric values of indices: $a = 2^0, b = 2^1, c = 2^2, d = 2^3$...
cd, abe,...	Variable combinations, these form the elements of an MBF.
$\emptyset$	The empty set
$\bot$	Bottom, this symbol represents the MBF of all zeros: {}
$\top$	Top, this symbol represents the MBF with all ones: {abc...}
D_α	The number of distinct MBFs in the equivalence class of α
$C_{\alpha,\beta}$	The number of connected components in the graph defined by $\alpha \setminus \beta$
$P_{n,2,\alpha,\gamma}$	P-Coëfficient for n variables, with 2 the jump size, and $\alpha \leq \gamma$ the Bounds. $P_{n,2,\alpha,\gamma} = 2^{C_{\alpha,\beta}}$

Chapter 1

Terminology

1. MBF: Monotonic Boolean Function: A boolean function which has the property of Monotonicity (See Section 3.2)
2. AntiChain: An AntiMonotonic boolean function (See Section 3.3)
3. Layer: A group of elements of a boolean function which have the same number of enabled variables, this refers to the lattice structure as shown in for example Figure 3.2.
4. Equivalence Class: the set of all MBFs that are equivalent to a given MBF up to permutation of the variables. Usually represented by one MBF from the class called the *Representative*.
5. Size Class: All MBFs that have a set number of enabled elements. There are $2^n + 1$ size classes in the lattice for $D(n)$
6. Domination: A term used for describing a 'greater than' notion. It's an operator that applies a partial ordering between MBFs and is described in Section 3.5.
7. Above and Below: This is a partial ordering relation between elements within an Boolean Function. The name comes from the visual structure of an MBF as shown in Figure 3.1. For example ac is above a and c, but not above b or ab or abc. ac is below abc, but not below bcd.
8. single-thread: Code that does not run in parallel
9. multi-thread: Code that runs in parallel. Multiple CPU cores may be executing code at the same time, and so it may make better use of the multiple cores in modern CPUs. This does introduce various issues such as requiring proper synchronization.
10. thread-safe: Operations on an object or container are either atomic, or locking is handled within the object's functions. No outside locking is required.
11. thread-compatible: Read operations on the object or container may happen in parallel, but writes must exclude all other operations first.

12. thread-unsafe: All operations (including read operations) must be protected by a lock. No two operations may happen at the same time.

13. Undefined Behaviour: A term for C++ programs that invoke erroneous operations, such as dereferencing a nullpointer, dividing by zero, or multi-threaded access of data without proper locking. It usually results in an error, but may exhibit quietly, producing incorrect results. It is to be avoided.

14. Mutex: Stands for 'Mutual Exclusion'. These are used for synchronisation in multithreaded programs. These allow the programmer to define blocks of code that only one thread can be executing in at a time.

15. Atomic: An atomic operation is one that is indivisible. It is safe to start multiple atomic operations at the same time. C++ defines `atomic<T>`: `atomic<T>` objects use atomic cpu instructions to perform thread-safe reads and writes to the object. These can be much more efficient than having to take a lock. This only works for register-size objects, such as integers and pointers. For larger objects, `std::atomic` has to use a `std::mutex` under the hood.

16. Lock-free: A C++ term describing a data structure which allows some multithreading capabilities without having to resort to full mutex locks. They are usually built using `atomic` components. Although very nice in theory, the constraints on the implementation can often be incredibly severe. Also, it is notoriously difficult to get the implementation right. These kinds of data structures are therefore only employed where they are absolutely necessary to reduce locking overhead.

Chapter 2

Introduction

The dedekind numbers are a hyper-exponential sequence of numbers representing the number of monotonic boolean functions with n variables, originally invisioned by Richard Dedekind in his book[1] in 1897. Computing these numbers is known as "Dedekind's Problem" and thus far only values from 0 up to and including 8 have been found (Table 2.1). It is conjectured that the complexity of computing these numbers is $O(2^{2^n})$, following the rising search space.

D(0)	2	Dedekind (1897)
D(1)	3	Dedekind (1897)
D(2)	6	Dedekind (1897)
D(3)	20	Dedekind (1897)
D(4)	168	Dedekind (1897)
D(5)	7581	Church (1940)
D(6)	7828354	Ward (1946)
D(7)	2414682040998	Church (1965)
D(8)	56130437228687557907788	Wiedemann (1991)

Table 2.1: Known Dedekind Numbers

R(0)	2
R(1)	3
R(2)	5
R(3)	10
R(4)	30
R(5)	210
R(6)	16353
R(7)	490013148

Table 2.2: Known Equivalence Class Counts

The last Dedekind Number that has been found, was D(8) in 1991 by Doug Wiedemann[2]. At the time, it took 200 hours on the Cray-2 supercomputer. Using

the exact same method he used yields time estimates for D(9) in the order of centuries on the largest supercomputers on earth. Even still, the construction of datasets about the structure of the 7-dimentional lattice, required for Wiedemann's method and related methods were still extremely expensive or downright unachieveable. In this thesis I present four new methods and advancements, that each surpass all known other methods.

1. Fast construction of all Equivalence Classes in the lattice of 7 Variables, and hence a recomputation of $R(7)$
2. For every equivalence class representative α, computation of the interval size $|[\bot, \alpha]|$
3. Optimizations to the P-Coëfficient method to reduce computationally expensive operations.
4. A theoretical FPGA acceleration that makes computation of D(9) feasible on currently available FPGA hardware.

The thesis is structured as follows: First there are two chapters with background, the first of which concerns the mathemathical objects and structures, the second talks broadly about C++ optimizing concepts. Chapter 5 lists the boolean function representation, as well as operations on it and their implementations that are used in later chapters. Emphasis is laid on the practical considerations for efficient implementation. Chapter 6 shows the first major algorithm for enumerating all Equivalence Classes in $R(7)$. Chapter 7 then builds a searchable data structure called the `EquivalenceClassMap` that saves properties for every equivalence class, and includes relations between Equivalence Classes that differ in size by 1. Chapter 8 uses the `EquivalenceClassMap` to compute all interval sizes $|[\bot, \alpha]|$ and adds these to the data structure. Everything then comes together in Chapter 9. Here we restructure the P-Coëfficient formula to use the `EquivalenceClassMap` to avoid some expensive operations, speeding up the computation tremendously. Finally, Chapter 10 theorizes about moving the inner loop of the P-Coëff formula to a dedicated piece of hardware implemented in FPGAs, and shows that if all goes well, it should be possible to compute D(9) in about 2 months.

2.1 C++ Code

All code developed in the context of this thesis is available on GitHub: `https://github.com/VonTum/Dedekind`. One may notice that there is some small overlap in code with another project of mine, named Physics3D[3]. The code that is shared is my custom testing framework, benchmarking framework, and some low-level cross-platform bitwise functions. I've also made use of a small library named uint256_t[4] for big integer representations. The code fragments in this text are written in the style of C++, but I took some liberties with regards to yielding functions, as this concept does not exist in C++, and the workaround makes the code quite unreadable.

Also type casts (such as casting a Boolean Function that has the structure of an AntiChain to an AntiChain) are left implicit.

The code was mostly developed and debugged in Visual Studio on Windows, but includes a CMakeLists.txt for GCC to compile for Linux required for supercomputing. It is structured into 5 projects:

1. **dedelib**: Compiles to a library, contains all dedekind-related code.
2. **indev**: Development testing and debugging
3. **production**: Commandline interface for generating all necessary files and running the Dedekind Number Computation
4. **tests**: Tests
5. **benchmarks**: Benchmarks

The **production** executable is then actually used to compute various datasets and dedekind numbers, either on a home computer, or the supercomputer for expensive operations. The possible commands are listed below. They are always structured as <command>N, with N the number of variables. So `./production genAllMBF7` would generate a binary file called `allUniqueMBF7.mbf` with all Equivalence classes in 7 variables.

1. **genAllMBF**: Generates all Equivalence Classes, stores them in `allUniqueMBF*.mbf` (∼10 minutes for N=7, 8GB on Disk)
2. **sortAndComputeLinks**: Generates all upLinks between all MBF Size Classes. Stores them in `allUniqueMBFSorted*.mbf` and `mbfLinks*.mbfLinks`. Requires `allUniqueMBF*.mbf`. (∼10 minutes for N=7, 20GB on Disk)
3. **linkCount**: Analysis of the generated links, produces a `linkStatsN.txt` file containing info and statistics. Requires `allUniqueMBFSorted*.mbf` and `mbfLinks*.mbfLinks`.
4. **computeIntervalsParallel**: Computes all $|[\bot, \alpha]|$ interval sizes and stores them in `allIntervals*.intervals`. Requires `allUniqueMBFSorted*.mbf`. (Heavy! ∼6.5 hours on 36-core supercomputer for N=7, 30GB RAM, 12GB on Disk)
5. **addSymmetriesToIntervalFile**: Adds Equivalence Class Sizes to a new file called `allIntervalSymmetries*.intervalSymmetries`. Requires `allIntervals*.intervals`. (∼1.5 hours for N=7, 20GB RAM, 12GB on Disk)
6. **computeDPlusOne**: Uses `allIntervalSymmetries*.intervalSymmetries` to compute D(N+1). Used as verification that the intervals are correct. (∼1-2 minutes for N=7, 20GB RAM)

7. **newPCoeff**: Uses `allIntervalSymmetries*.intervalSymmetries` to compute D(N+2) using the P-Coëfficient method. (∼6 minutes for N=6. Centuries for N=7. 60GB RAM)

8. **benchmarkSample**: Generates a benchmark sample `benchmarkSet*.intervalSymmetries` to be used in benchmarks. Requires `allIntervalSymmetries*.intervalSymmetries` (∼1 minute for N=7, 20GB RAM, 0.1GB on Disk)

9. **benchmarkSampleTopBots**: Generates a benchmark sample `benchmarkSet*.topBots` to be used in benchmarks. Requires `allIntervalSymmetries*.intervalSymmetries` (Heavy! ∼22 hours on 36-core supercomputer for N=7, 60GB RAM, 7GB on Disk)

All executables also provide a `--dataDir <Dir>` flag for setting the *data* directory where all these files are stored. by default this is `./data/`.
Everything combined it's about 14'000 lines of code. The generated data sets can be provided upon request.

Chapter 3

Literature Background

This thesis builds upon the work of Patrick De Causmaecker, Stefan De Wannemacker, and Jay Yellen. Using concepts from their papers "Partitioning in the space of antimonotonic functions"[5], "On the number of antichains of sets in a finite univese"[6], and "Intervals of Antichains and Their Decompositions"[7]. In their papers they mostly deal with AntiChains. In this thesis instead I will mostly be talking about Monotonic Boolean Functions (MBFs). AntiChains and MBFs map one-to-one and have similar properties.

3.1 Boolean Functions

The boolean function is a simple primitive with a number of boolean inputs (called Variables) and a boolean output. Simple examples of boolean functions are AND, returning *True* only when both inputs are *True* and OR, returning *True* when at least one input is *True*. They can be generalized as a table of 2^N output values, each of which correspond to a specific set of inputs. Another representation of these Boolean Functions is as a set of sets. Each of the smaller sets corresponds to a '1' entry in the table representation, and contains the inputs that were '1'. As an example, the boolean function shown in Table 3.1 could also be represented as $\{\{\}, \{b\}, \{ab\}, \{c\}, \{abc\}\}$. Therefore we can refer to the inputs for which the boolean function returns '1' as the 'elements' of the boolean function. This also gives rise to the concept of one boolean function being a 'subset' of another. One boolean function is a subset of another if for every input that the first boolean function returns '1', the other also returns '1'. Finally, there is a relation between the elements of a boolean function that is used to define monotonicity called 'domination'[5], though I will refer to it as one element being 'above', or being 'below' another[1]. An element (such as $\{ab\}$) is 'above' another (such as $\{b\}$) if the variables that appear in the second are a subset of those that appear in the first. Likewise, $\{acd\}$ is 'below' $\{acde\}$ because every variable that appears in $\{acd\}$ also appears in $\{acde\}$. This is expressed using the $\leq$ operator, so $\{acd\} \leq \{acde\}$

[1] This notation refers to their visual relation, as shown in Figure 3.1

inputs: c,b,a	output
000	1
001	0
010	1
011	1
100	1
101	0
110	0
111	1

Table 3.1: An example boolean function in table representation. Entries are marked by letters which signify the inputs for which the function gives the given output. In this example the input {a=1,b=1,c=0} corresponds to the *ab* field, and the function returns *True*. For {a=1,b=0,c=1} the function returns *False*

3.2 Monotonic Boolean Functions

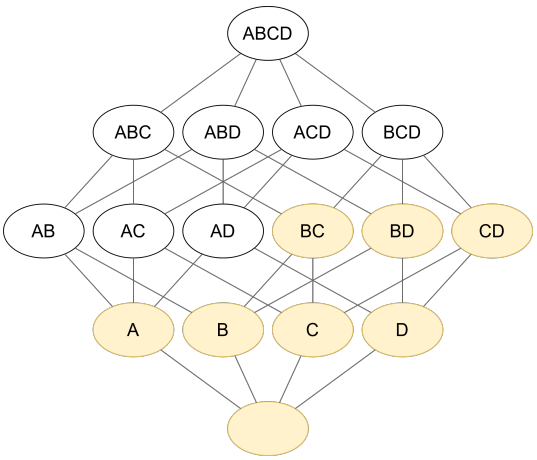

Figure 3.1: An example Monotonic Boolean Function. The yellow nodes return *True*, and the white nodes return *False*. Nodes below yellow nodes must be yellow. Nodes above white nodes must be white.

Monotonic Boolean Functions (MBFs for short) are Boolean Functions where the following condition applies: If the function returns *True* for a given input (also called element), then any input that is a subset of this input must also return *True*. Equivalently, the function returning *False* for a set of inputs requires that the function also return *False* for any superset of this input.
In set notation, if the boolean function contains a set ($\{ab\}$), then it must also contain all subsets of that set. ($\{ab, a, b, \emptyset\}$)
As an example the Boolean Function in Table 3.1 is not monotonic, since $a \leq ab$, the function returns *True* for *ab* but not for *a*. The Boolean Function shown in Figure 3.1 is monotonic.

Mathemathically this is expressed as:

$$\gamma \textit{ is monotonic} \iff \forall x \in \gamma : \forall y \subseteq x : y \in \gamma \tag{3.1}$$

Some authors reverse the definition of Monotonic Boolean Functions, going from *False* (for *False* inputs) to *True* (for *True* inputs). It does not matter which one, one picks, the underlying structure remains the same. In this thesis Monotonic boolean functions are defined as going from *True* to *False*. This makes conversion between AntiChains and MBFs easier, and allows Meets and Joins to mirror AND and OR.

3.2.1 Layers

A Boolean Function Layer is a group of elements of a boolean function which have the same number of enabled variables, this refers to the lattice structure as shown in for example Figure 3.1, the MBF Layers are the horizontal layers. Layers are also AntiChains by construction. 'Layer' may refer to the whole set of elements with the same number of variables, or to a subset, depending on the context.

3.3 AntiChains

AntiChains (ACs) are kind of the opposite of MBFs, they are Boolean Functions where if an element is *True*, then no element above or below by it may be *True*. An example is shown in Figure 3.2 (Right)
Mathemathically this is expressed as

$$\gamma \in A_n \iff \forall x \in \gamma : \neg(\exists y \in \gamma : y \subset x) \tag{3.2}$$

3.4 Monotonic - AntiChain duality

There is a duality between Monotonic Boolean Functions and AntiChains. Any AntiChain can be converted to an MBF by including all the elements below any of the AntiChain elements. Likewise any MBF can be converted to an AntiChain by removing any elements that is below another. This is shown in Figure 3.2. Though these are functionally identical, various operations can be implemented more efficiently in one representation over the other. When noting down an MBF in text form, we usually write it as an AntiChain to reduce writing overhead.

3.5 Partial ordering on MBFs

We'll define a partial ordering over the MBFs, using which various theorems and properties can be proved. Many of these properties over the AntiChain equivalent of this partial ordering are explored in [5]. Since we are working mostly with MBFs, I give an equivalent definition of this ordering over MBFs:
One MBF is 'smaller than' another if it is a subset of it.

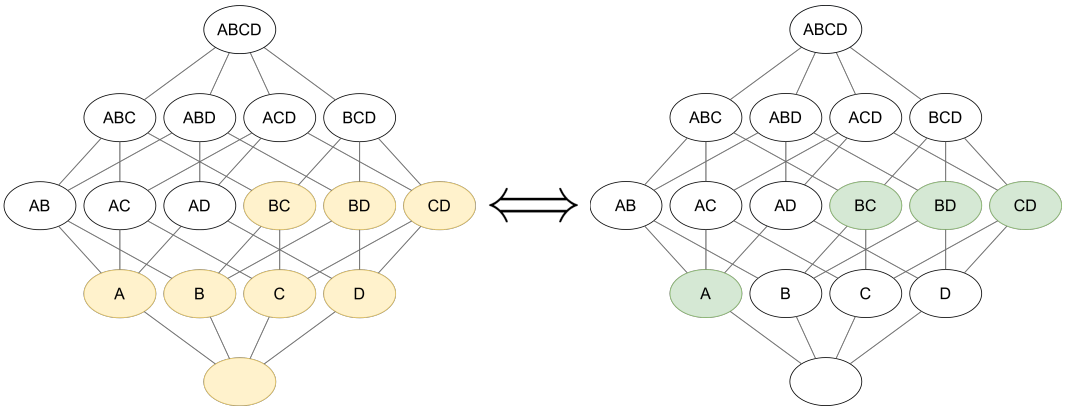

Figure 3.2: The duality between an example MBF (Left) and it's AntiChain counterpart (right) $\{a, bc, bd, cd\}$

With this ordering there are two special MBFs that are given a name, Top $\top$ and Bottom $\bot$. $\top$ is the MBF that returns *True* everywhere, so every other MBF is a subset of it. $\bot$ returns *False* everywhere, so it is a subset of every other MBF.

3.6 Joins, Meets, and the Distributive Lattice of MBFs

Between two MBFs (or AntiChains), a *Join*, and a *Meet* operation can be defined. These operators are defined in [5].
The *join* is the smallest MBF (compared to all others) that is larger than the two given MBFs:

$$\gamma = \alpha \vee \beta \iff (\alpha \leq \gamma)\ \&\ (\beta \leq \gamma)\ \&\ (!\exists \delta \neq \gamma : (\alpha \leq \delta)\ \&\ (\beta \leq \delta)\ \&\ (\delta \leq \gamma)) \quad (3.3)$$

Similarly, the *meet* is defined as the largest MBF that is smaller than the two MBFs.

$$\gamma = \alpha \wedge \beta \iff (\gamma \leq \alpha)\ \&\ (\gamma \leq \beta)\ \&\ (!\exists \delta \neq \gamma : (\delta \leq \alpha)\ \&\ (\delta \leq \beta)\ \&\ (\gamma \leq \delta)) \quad (3.4)$$

Meets and Joins are unique, and therefore well-defined. This is required for the formation of a lattice. All MBFs can be arranged into a lattice over the Join and Meet operators. With the joins going up towards $\top$ and meets going down towards $\bot$. The size of this lattice is equal to the N-th dedekind number, with N the number of variables. Example these lattices and their sizes are shown in Figure 3.3

3.7 Intervals

Another important concept is the *Interval*. It represents the set of MBFs that lie between two MBFs. Written as

$$\forall \alpha, \beta, \gamma \in MBF_n : \gamma \in [\alpha, \beta] \iff (\alpha \leq \gamma)\ \&\ (\gamma \leq \beta) \quad (3.5)$$

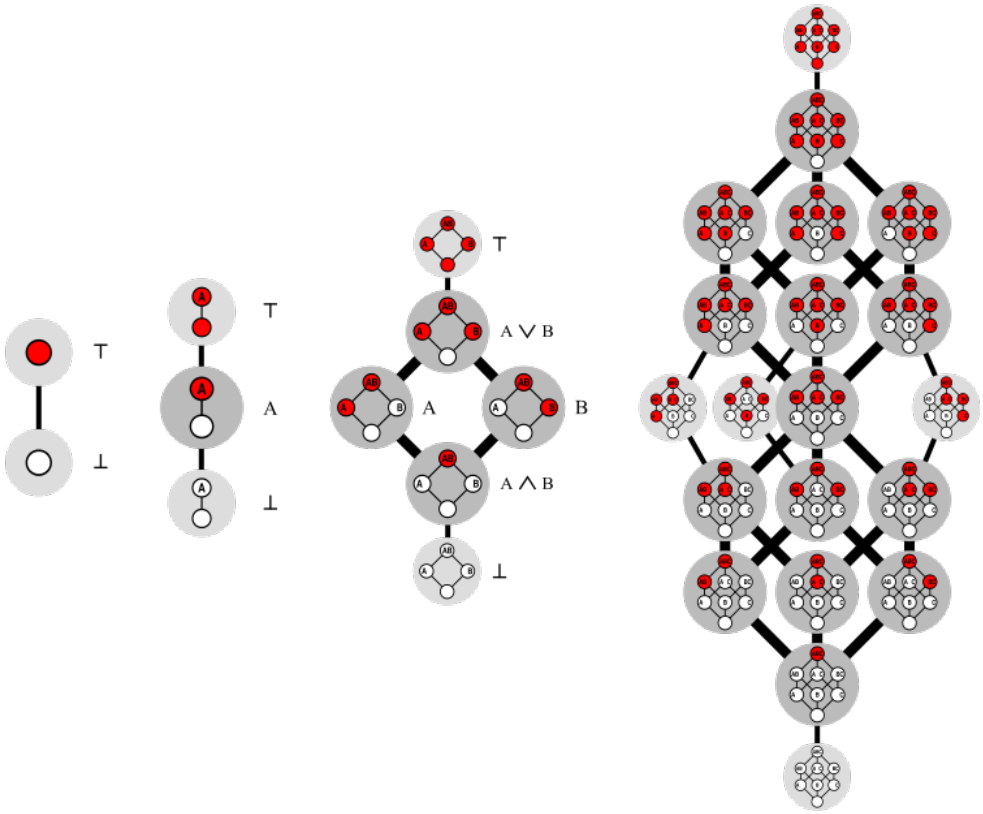

Figure 3.3: The lattices for N=0,1,2,3, joins go up towards $\top$, meets go down towards *bot*. In this image from Wikipedia the monotonic representation is inverted, functions go from *False* for $\emptyset$ to *True* for $abc\ldots$. This presents the Dedekind numbers 2, 3, 6, and 20 respectively. Taken from Wikipedia (Creative Commons)[8]

There are a whole bunch of properties and operators on intervals that are explored in [5], but as we do not need them in this thesis, they are omitted. Only a few simple properties are used. Those are shown in Section 5.5.
An example of the interval between a and ab, c is shown in Figure 3.4.

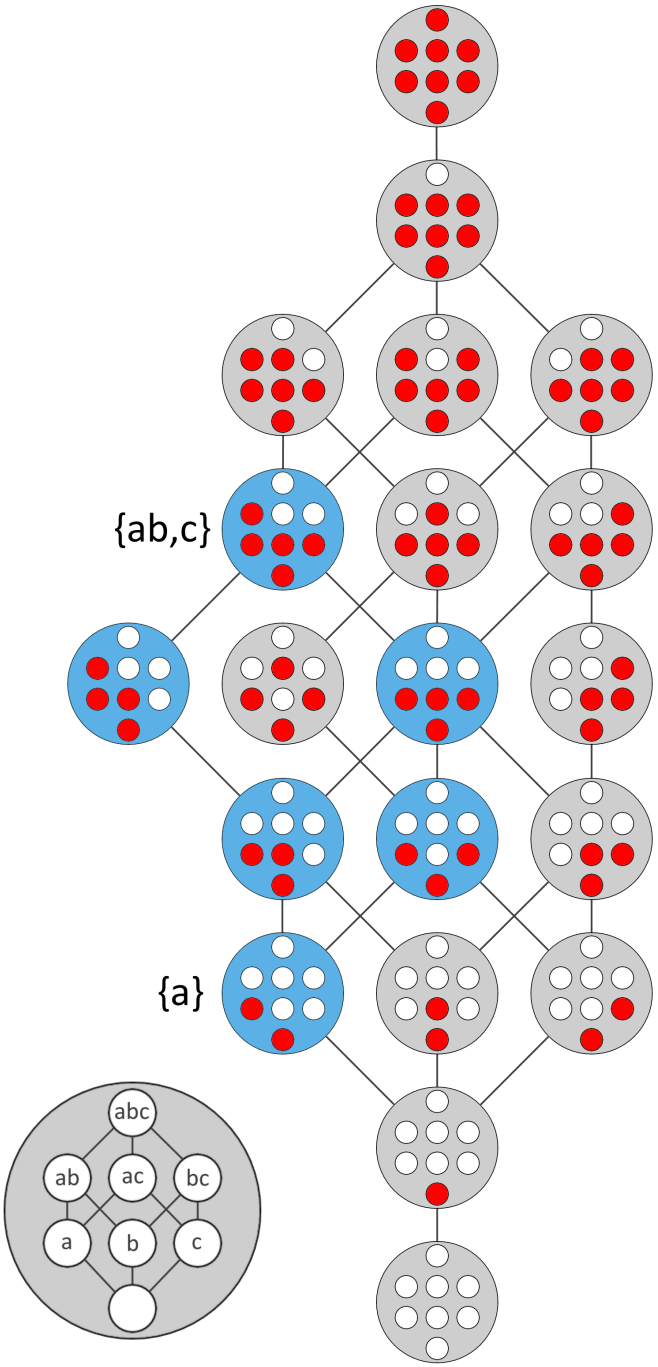

Figure 3.4: An example interval is shown here. All MBFs marked in blue are contained in the interval $[\{a\}, \{ab, c\}]$

Chapter 4

Programming Background

Special attention has to be given to the various techniques that exist to optimize and profile code, to allow us to use the available hardware as efficiently as possible. First and foremost is the choice of programming language. There are few modern languages which still compile directly to executable assembly, this is important, as (semi-)interpreted languages such as Java or Python that provide a higher abstraction over the hardware won't allow us to make the low-level optimizations we need. The language of choice for this thesis is C++. Here are some arguments as to why: First of all, C++ is one of the most widely used languages in the industry, and due to this, the various tools and compilers that exist currently (mainly GCC, clang, MSVC, and icc) have seen decades of improvements, and as such produce incredibly efficient assembly code. These compilers are so clever that they will even produce different assembly for different CPU models even when they use the exact same instruction set, just to cope with minor inefficiencies within the specific CPU models. Simply transcribing java code to C++ can easily yield a 2-5x performance gain[9], and further gains are usually possible with manual optimization. Secondly, the C++ standard library allows for working at high levels of abstraction using the builtin types, while also allowing the programmer to get into the very low level implementation details, allowing us to even write assembly directly where necessary. Finally there is an ecosystem of tools, resources and talks specifically for optimizing C++ code. Visual Studio [10] and especially its profiler were invaluable in finding and optimizing the bottlenecks in the code. Matt Godbolt's Compiler Explorer [11] is also really useful for understanding how the compiler generates assembly code, what things compilers can and can't optimize, and making sure that all compilers generate similar assembly code.

Every year there are several C++ conventions, where members of the C++ committee and experts in the field come to talk about topics such as code cleanliness, testing, profiling, and performance. A sample of performance-oriented talks is cited here, though there are many more. There are talks with generic optimization guidelines [12][13][14] while other talks go into detail on specific topics, such as data structures [15], memory allocation [16], caching [17], threading and locking [18], and atomics [19][20]. Some talks go even deeper, really thinking about the specific

hardware one has available. They talk about SIMD, pipelining, branch prediction and register/execution unit pressure. [21]
I mostly leave the specific optimization details out of the text as that would lead us way too far into the nitty gritty implementation details. Though I do discuss parallelization, memory alignment and caching in Chapters 6 and 7 since these play a significant role there. In the other chapters, parallelization is trivial to implement. SIMD is also mentioned specifically in Chapter 8 as this is where it was used to the greatest effect.

4.1 SIMD Instructions

SIMD stands for "Single Instruction (stream) Multiple Data (stream)". It allows performing instruction-level parallelism. Using these instructions one can do the same operation on 4, 8, or 16 data streams at once, greatly boosting performance. On the x86 platform, there are several SIMD instruction set extentions that have been built over the years.

Shorthand	**Name**	**Register Width**
MMX	MultiMedia Extentions	64
SSE	Streaming SIMD Extentions	128
AVX	Advanced Vector Extentions	256
AVX-512	Advanced Vector Extentions-512	512

Table 4.1: x86 SIMD instruction sets by Intel

SIMD vector registers pack a number of values together in one register, for example AVX can fit 8 32-bit floats in its `__m256` registers. Then there are instructions which take these SIMD registers and perform 4, 8, or 16 operations in parallel. As an example `__m256 _mm256_mul_ps (__m256 a, __m256 b)` multiplies 8 pairs of floats together and returns 8 floats. The full list of x86 intrinsics is available at the Intel Intrinsics Guide[22]. While AVX is quite widely supported on consumer hardware, AVX-512 has been mostly limited to Intel's Server-Oriented CPUs.

Chapter 5

Overview of Operations

5.1 Boolean Function Representation

The underlying representation for boolean functions is a *BitSet*. Boolean Functions of N variables are represented as a bitset of length 2^N, where every bit gives the result of the function for the input that is encoded in the index of the bit. This can be implemented as a fixed-size bitset. This means all operations are executed on the underlying integer of the whole bitset. Many operations will map onto boolean operations and shifts.

0	1	0	1	1	1	0	1
abc	bc	ac	c	ab	b	a	{}
111	110	101	100	011	010	001	000

Table 5.1: Example Boolean Function Bitset Representation. The top row is the bitset, below it are the active variables of each element, and the bottom row represents the presence as binary vectors, these also represent the indices in the bitset. This example is not monotonic because it includes *ab*, but not *a*

The position of every element in this representation is given by:

$$\sum_{v}^{Vars} 2^v \tag{5.1}$$

So for example if we wish to know the index of *acd*, we calculate $2^0 + 2^2 + 2^3 = 1 + 4 + 8 = 13$.
Using this kind of structure allows us to efficiently express the adding and removing of variables as shifts. To add the variable c to some elements that don't contain c we can simply shift the whole bitset by $2^2 = 4$. Shifting the bits by x is like adding x to the index of every element, which is exactly what we want to do when adding an additional variable to all elements.

5.2 Operations applicable to any Boolean Function

The performance of operations on this representation is critical, as this determines the performance of the whole application. Luckily, with this representation the operations we need can be implemented efficiently, often as single assembly instructions. A summary of Boolean Function operators and functions is listed below, the implementation of each will then be explored in its own section below.

1. $\&, |, \hat{}, \neg$ - Bitwise Boolean operations
2. $\leq_\Omega$ - Total ordering
3. $varMove$ - Variable Moving
4. $varSwap$ - Variable Swapping
5. Permutations of the variables
6. $monotonizeDown$ - Add all elements that are dominated by at least one element in the BF
7. $monotonizeUp$ - Add all elements that dominate at least one element in the BF
8. $|*|$ - Population Count (Count number of 1s)
9. $first$ - Find first (lowest order) '1' bit
10. $last$ - Find last (highest order) '1' bit
11. $powerset$ - Powerset iteration
12. Canonisation
13. $\simeq$ - equivalence up to permutation

5.2.1 Bitwise Boolean Operators

The basic bitwise operators can also be applied to bitsets, and therefore also on Boolean Functions, as they are built on uint64 integers under the hood.

5.2.2 Total Ordering

There is no real mathemathical meaning behind this specific ordering, as it is arbitrary, but defining a total order over all Boolean Functions (and by extention Monotonics and AntiChains) is quite useful for canonization or deduplication. As well as when using data structures that employ binary searches.
For my implementation I defined my total order as unsigned integer comparison on the underlying data of the bitset.

5.2.3 varMove

varMove renames a variable. Every occurence of x should be replaced with y. The target variable should not have been used yet to get a consistent result. $varMove$ is mostly just a precursor to *varSwap*.

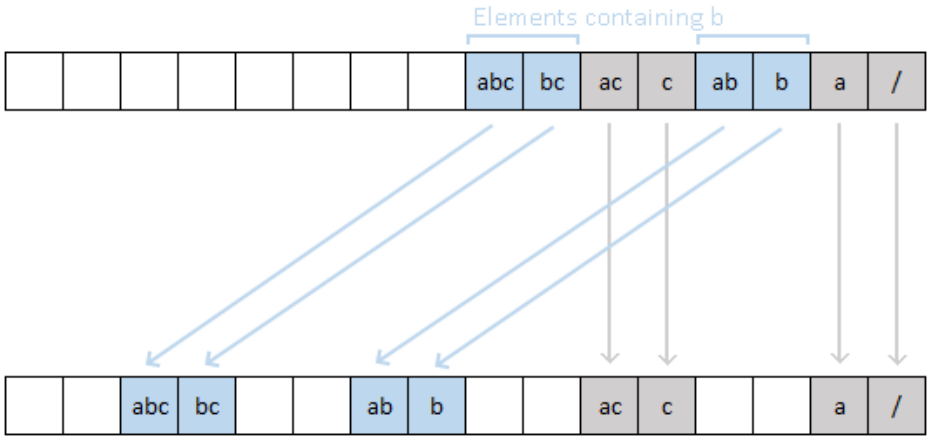

Figure 5.1: Moving variable b to slot d. Notice d is not used yet (represented by empty blocks). After the operation, the slots of former b elements are now unused. In this case the shift is $2^3 - 2^1 = 6$

Say for example this set contains x, to rename it to y, we can simply remove the 2^x and add 2^y. Since we add the same amount for every bit, we can map it nicely to a bitshift of the whole set. We must make sure to only shift the entries containing x though, so we first filter those out with a bitmask and then merge the shifted and unshifted results together as shown in the code.

```
BF varMasks[]{
  0b10101010101010101010101010101010..., // a
  0b11001100110011001100110011001100..., // b
  0b11110000111100001111000011110000..., // c
  0b11111111000000001111111100000000..., // d
  0b11111111111111110000000000000000..., // e
  // etc...
}
// bf does not contain an element where y is active
BF varMove(BF bf, int x, int y) {
  BF containingX = bf & varMasks[x];
  BF notContainingX = bf & ¬varMasks[x];
  int shift = 2^y − 2^x;
  return (containingX << shift) | notContainingX;[a]
}
```

[a] negative shifts are interpreted as shifts in the other direction.

5.2.4 varSwap

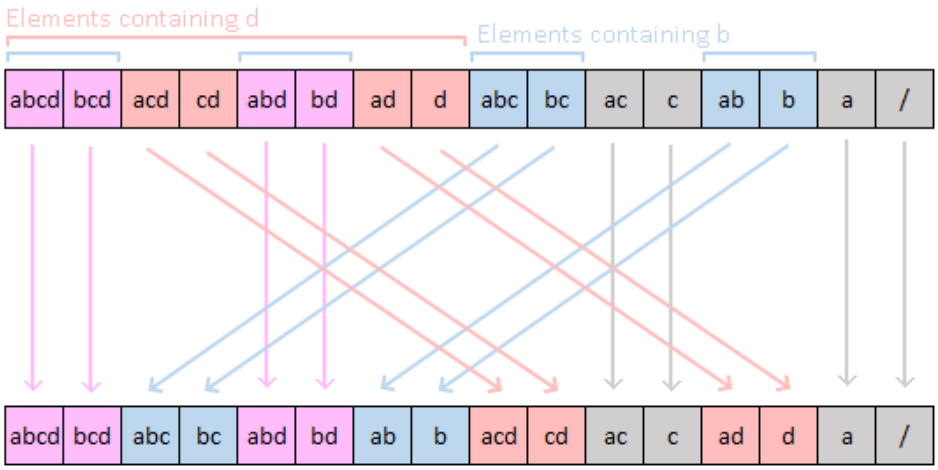

Figure 5.2: Swapping b and d in a 4-variable Boolean Function.

varSwap expands upon *varMove* by swapping the two variables. This also removes the need for the target variable to be unused. It works by noticing that to remove a variable x and add a variable y one can simply add $2^y - 2^x$, and vice verca. This translated to shifts masking out and shifting the elements containing only one of x or y. Elements containing neither or both stay in place.

```
BF varSwap(BF bf, int x, int y) {
  BF maskX = varMask[x];
  BF maskY = varMask[y];
  BF staying = bf & (maskX & maskY | ¬maskX & ¬maskY);
  int shift = 2^y − 2^x;
  BF xToY = bf & (maskX & ¬maskY);
  BF yToX = bf & (¬maskX & maskY);
  return staying | (xToY << shift) | (yToX >> shift);
}
```

5.2.5 Permutations

Once we can swap arbitrary variables, we can start performing arbitrary permutations as well by applying multiple swaps. A permutation p of a Boolean Function α is written as $p(\alpha)$, and can be split into a permutation of the elements:

$$p(\{x, y, ...\}) = \{p(x), p(y), ...\} \tag{5.2}$$

Permutations are a bijection, meaning that a permutation can't map two different elements onto the same permuted element:

$$x \neq y \iff p(x) \neq p(y) \tag{5.3}$$

. This also means that permutations are invertible, so every permutation p has an inverse permutation $\overline{p}$, for which:

$$p(\overline{p}(\alpha)) = \overline{p}(p(\alpha)) = \alpha \tag{5.4}$$

5.2.6 monotonizeDown and monotonizeUp

Monotonization is used to convert AntiChains to Monotonics, but it also sees use in later algorithms. *monotonizeDown* takes a boolean function, and enables all elements that are dominated by enabled elements of the input. This can be implemented efficiently using boolean logic, by cumulatively removing one element at a time. This way the parallelism inherent in binary operations can be exploited to the fullest. *monotonizeUp* is very similar to *monotonizeDown* where the only differences are that shifts are reversed and the masks are inverted.

```
BF monotonizeDown(BF bf) {
  BF result = bf;
  for(int v = 0; v < Variables; v++) {
    result = result | ((result & varMask[v]) >> 2^v);
  }
  return result;
}
```

5.2.7 Built-in Instructions: popcount, first and last

These operations are notoriously difficult to implement in code, which is why modern CPU manufacturers have implemented dedicated CPU instructions for these operations. x86-64 defines the following instructions:

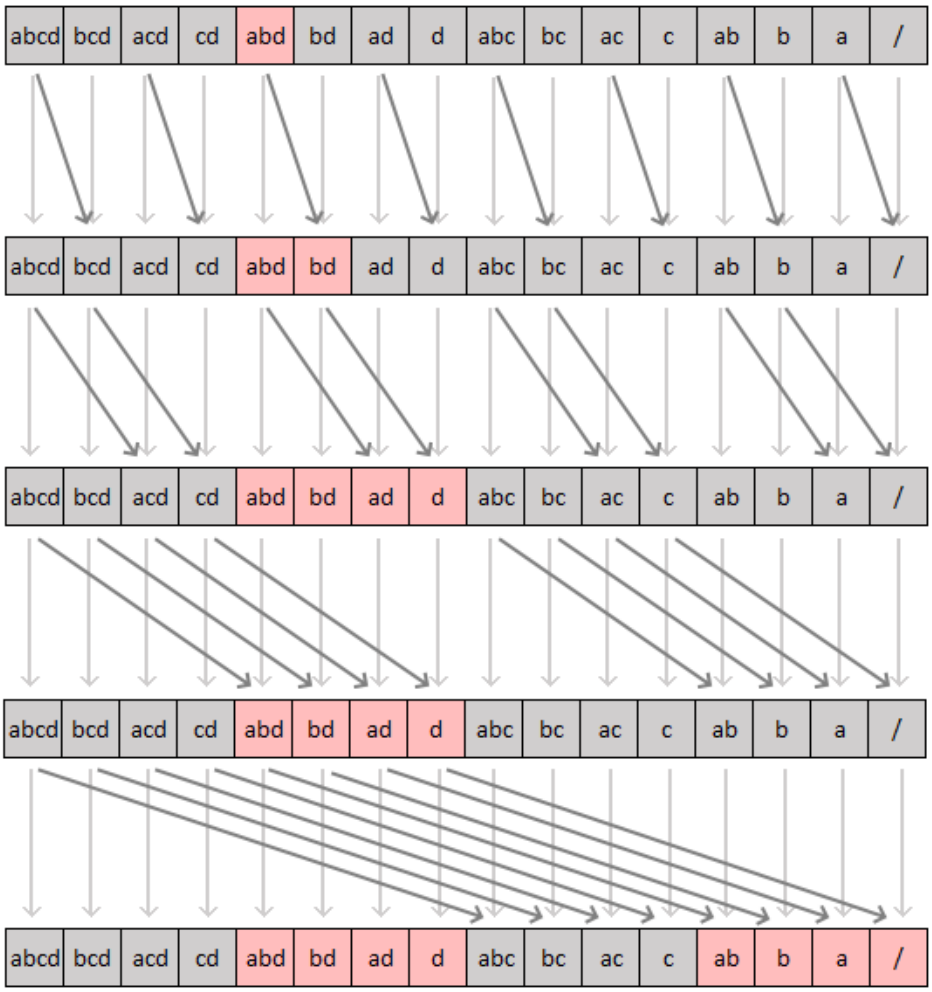

Figure 5.3: *monotonizeDown* for 4 variables. Example monotonization of $\{abd\}$. Red squares signify '1' elements, gray squares are '0'. After monotonization any element dominated by $\{abd\}$ is now '1'

1. `popcnt` - Population count: Count number of '1' bits
2. `bsf` - BitScanForward: Get index of lowest order '1' bit, undefined for 0
3. `bsr` - BitScanReverse: Get index of highest order '1' bit, undefined for 0

These only go up to 64 bits, but we can build larger bitwidth versions just by combining multiple instructions. (eg summing two `popcnt`s for a 128-bit `popcnt`, or checking for zeros first for the bitscan equivalents.) Using these instructions it's possible to implement $popcount$,$first$, and $last$.
GCC (and compatible compilers) and MSVC both provide intrinsic functions to avoid having to write assembly oneself.

5.2.8 powerset

Iterating over the powerset of a set is used in a few operations involving AntiChains, though it can be applied generally, so I include it in this section. We wish to iterate over all possible subsets of a set of '1' bits in a given bitset. There's a really neat way to do this using integer decrement rollover. If the bits were arranged nicely at the end of the bitset, then we could just decrement the integer representation of our bitset until we hit 0: 0b111->0b110->0b101->0b100->0b011->0b010->0b001->0b000. While this doesn't work generally, we can modify this slightly to make it work in the general case by masking out the bits that get messed up outside of the elements we wish to iterate over.

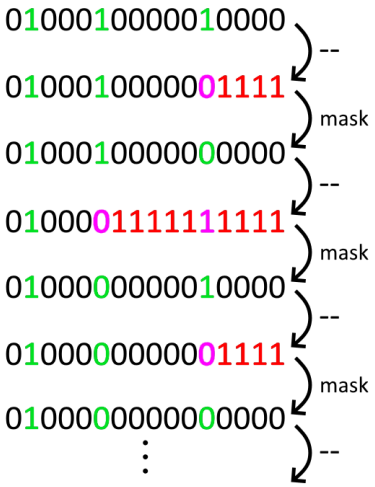

Figure 5.4: Powerset iteration example. The bitstrings in green and black are the generated subsets.

Sample code:

```
generates BF powerset(BF mask) {
  BF cur = mask;
  do {
    yield cur;
    bf--;
    bf &= mask;
  } while(cur != mask);
}
```

5.2.9 Canonisation and Equivalence Classes

In many algorithms related to Monotonic Boolean Functions, there are symmetries that can be exploited. Many properties are preserved under permutation of the variables, so it's often useful to only compute these properties once on one MBF, and reuse them for all permutations of it. It's best to look at the (hyper-) graph representation of Boolean Functions for this. We define every variable as a node, and any enabled element in the Boolean Function corresponds to a hyperedge between

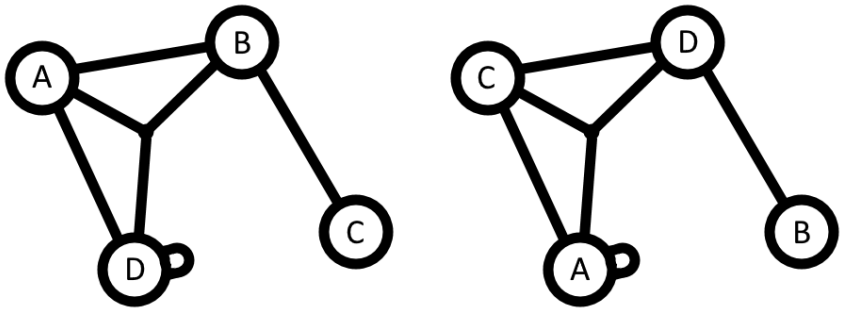

Figure 5.5: Example hypergraph representations of two Boolean Functions which are equivalent under permutations of the variables. Shown left is a 3-way hyperedge between ABD, and a 1-way hyperedge of just D. This does not represent a monotonic boolean function.

the involved nodes. Have a look at Figure 5.5: the hypergraphs aren't the same, the vertices are labeled differently, but the underlying structure of the hypergraph is the same. Many properties that hold for a specific naming of the vertices will hold for any permutation of the variables. For example, the number of enabled elements, or the interval size to top or bottom. To this end we define *Equivalence Classes*, these represent the structure of the graph/boolean function rather than a specific naming of the vertices, so the two graphs in the example would belong to one Equivalence Class. The operator we use to denote that two Boolean Functions belong to the same Equivalence Class is $\simeq$.
To represent such an equivalence class we pick one instance in the class and call it the representative of the class. We also define a function (called the canonization function) that converts any elements in the class to this representative. One simple way to do this is to make use of the arbitrary total ordering relation we defined earlier. Using the fact that one element will be the largest (per the well ordering principle) we can designate this element as the class representative. To do this for an arbitrary boolean function we can iterate over all the elements in its equivalence class by iterating over all possible permutations of its variables and finding the maximum. This may seem to have a scary $O(n!)$ complexity, but remember that we are talking about graphs with very few vertices here, so the actual time stays relatively small.
Still, we can improve this, the only requirement of canonization is that all permutations map to the same instance, regardless of which it specifically is. Hence, we can reduce our search space significantly by forcing a partial order on the variables/vertices based on properties not affected by permutation, such as the number of incoming hyperedges in each vertex, or even more specifically, the number of incoming hyperedges per arity. This will split our vertices into several distinct groups, we can deterministically assign variable groups to these subgroups using some other arbitrary ordering (for example ordering by largest count of largest hyperedge first). In the example shown in Figure 5.6 this reduces the search space from 4! to $1! * 1! * 2!$.
We can go even further, once we have a basic grouping, we can refine within these groups by looking at each vertex' connections to other groups. This refinement can

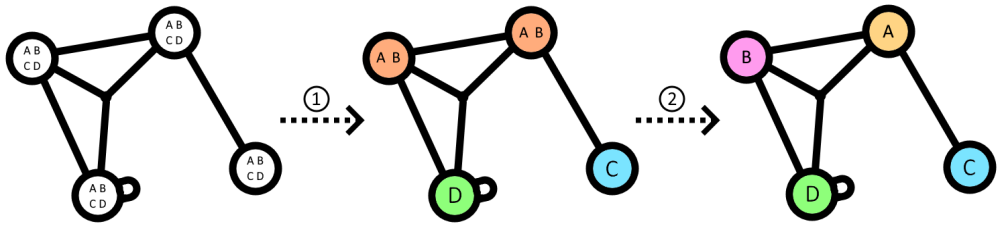

Figure 5.6: Progression of vertex distinguishing. Possible names for the vertices are placed within each vertex.
(1) shows initial groups from counting incoming edges.
(2) shows refined groups from group connections.

be repeated until no new groups are created. In the example, A and B couldn't be distinguished previously as they both have the same number of connections, but now that connections are counted per group, we are able to distinguish them. In this example, every vertex is distinguishable, so we immediately know the canonical order of the variables, but that is not always the case. If we are still left with groups of size >= 1 then we must fall back on permutation iteration within the groups. Still though these reduced intra-group permutations are a massive improvement over the previous full $n!$ permutation.
Performance measurements are listed in Table 5.2.

Canonization algorithm	**Boolean Function**	**Monotonic**
Full permutation	11500ns	11500ns
Basic grouping w/o refining	320ns	388ns
Grouping with refining	260ns	333ns

Table 5.2: Time taken per canonization of a boolean function with 7 variables with various methods. Averaged over 1000 iterations where per iteration every permutation of a random (monotonic) boolean function is canonized.

5.3 Efficient operations for MBFs

Over MBFs (and by extention AntiChains) we can define several operators which help us move around the MBF lattice. These operators are summarized here, and explained in more detail in the sections below.

1. $\leq$ - MBF comparison
2. $\vee$ - Join
3. $\wedge$ - Meet
4. *pred* - Predecessor

5. *succ* - Successor

6. $\overline{\alpha}$ - Dual

5.3.1 Partial Order, Joins and Meets

For our MBF representation, these can all be implemented as binary logic. The partial ordering is just a subset check.

$$a \leq b \iff a \,\&\, b = a \tag{5.5}$$

Join and meet are implemented as boolean `OR` and `AND`:

$$\alpha \vee \beta = \alpha | \beta \tag{5.6}$$

$$\alpha \wedge \beta = \alpha \,\&\, \beta \tag{5.7}$$

5.3.2 Predecessor and Successor

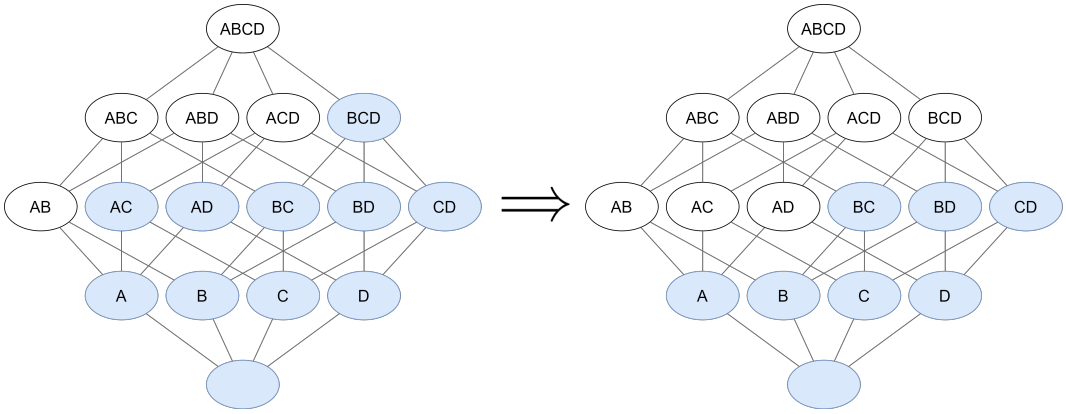

Figure 5.7: Predecessor

AntiChain definitions for these can be found in [5]. These are the equivalent definitions for MBFs:

1. The predecessor of an MBF α removes all elements from α for which no element, extending it with one variable, exists. Visually this is removing all elements for which all elements above are *False*, as shown in Figure 5.7.

2. The successor of an MBF α adds all elements for which all elements that have one variable removed are in α. Visually this is adding new elements for which all elements below are *True*, as shown in Figure 5.8.

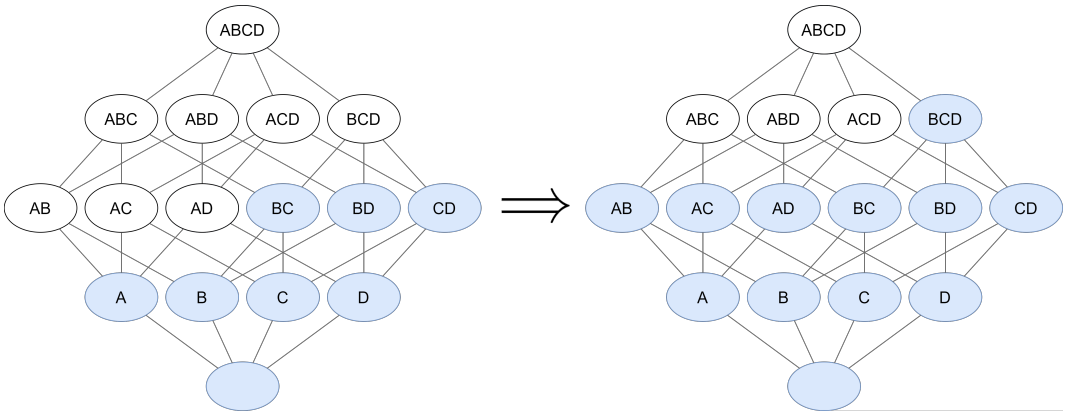

Figure 5.8: Successor

5.3.3 Dual

Every MBF has a dual MBF, this dual is a kind of 'inverse' of the MBF. An element is in the dual of γ if the inverse element is not in γ:

$$\forall x : x \in \overline{\alpha} \iff \overline{x} \notin \alpha \tag{5.8}$$

With $\overline{x}$ the bitwise inverse of x, meaning all variables in x are not in $\overline{x}$
In other words, to find the dual of an MBF, you invert the inputs, and you invert the output.

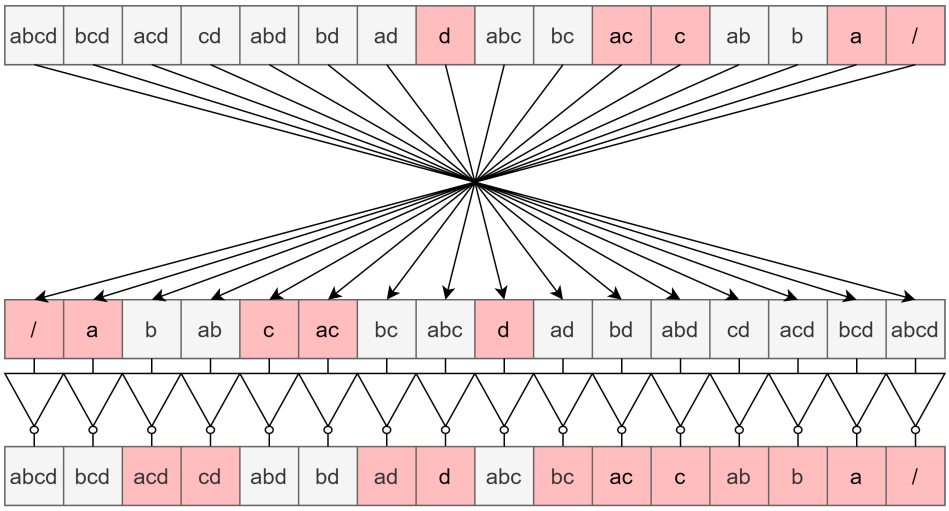

Figure 5.9: Taking the dual of $\{d, ac\}$ gives $\{acd, bc, ab\}$

Properties of duals include:

1. The dual of the dual is the identity. $\overline{\overline{\alpha}} = \alpha$
2. The dual is unique. $\overline{\alpha} = \overline{\beta} \iff \alpha = \beta$
3. MBF comparison is reversed: $\alpha \leq \beta \iff \overline{\beta} \leq \overline{\alpha}$
4. Equivalence Class duality: all monotonics in an equivalence class map to distinct MBFs in the dual Equivalence Class. This directly leads to the next point:
5. Dual Equivalence Class size: The size of the equivalence class of the dual of an MBF is the same as the Equivalence Class size of the MBF. This follows from 2 and 4.
6. Interval size of the dual interval are preserved: $|[\alpha, \beta]| = |[\overline{\beta}, \overline{\alpha}]|$. To prove this, we apply reversing of MBF comparison: $\forall \alpha, \beta, \gamma \in A_n : \gamma \in [\alpha, \beta] \iff \alpha \leq \gamma \leq \beta \iff \overline{\beta} \leq \overline{\gamma} \leq \overline{\alpha} \iff \overline{\gamma} \in [\overline{\beta}, \overline{\alpha}]$, and because each γ corresponds to a unique $\overline{\gamma}$, $|[\alpha, \beta]| = |[\overline{\beta}, \overline{\alpha}]|$. □

5.4 MBF - AntiChain Conversion

Because both representations are required for an efficient implementation, fast conversion functions are a necessity. Luckily such conversions are quite easy to implement in this representation.

1. AntiChain ⇒ Monotonic: To convert an AntiChain to an MBF, one can simply perform a *monotonizeDown* operation.
2. Monotonic ⇒ AntiChain: To convert an MBF to an AntiChain, we must eliminate all elements that have another element above them. We can get these elements using the *predecessor* operation. To remove them it's just a boolean `AND NOT`: $AC = MBF \ \& \ \neg pred(MBF)$

5.5 Intervals

Intervals represent subsets of the whole D(N) space. They are defined as MBFs that are larger than the bottom MBF of the interval, and smaller than the top MBF.

$$\gamma \in [\alpha, \beta] \iff \alpha \leq \gamma \ \& \ \gamma \leq \beta \tag{5.9}$$

Intervals that are of specific interest to this thesis are intervals of the form $[\bot, \beta]$, the condition then simplifies to:

$$\gamma \in [\bot, \beta] \iff \gamma \leq \beta \tag{5.10}$$

An important property of intervals is taking their size:

$$|[\alpha, \beta]| = \sum_{\substack{\alpha \leq \gamma \\ \gamma \leq \beta}} 1 \tag{5.11}$$

For our intervals starting from bottom this simplifies to:

$$|[\bot, \beta]| = \sum_{\gamma \leq \beta} 1 \tag{5.12}$$

A property of intervals and their sizes is that they are not affected by global permutation. So given a permutation p this gives:

$$\forall \texttt{ permutation } p : \gamma \in [\alpha, \beta] \iff p(\gamma) \in [p(\alpha), p(\beta)] \tag{5.13}$$

$$\forall \texttt{ permutation } p : |[\alpha, \beta]| = |[p(\alpha), p(\beta)]| \tag{5.14}$$

Intervals and their sizes are also preserved over taking the dual of all involved MBFs:

$$\gamma \in [\alpha, \beta] \iff \overline{\gamma} \in [\overline{\beta}, \overline{\alpha}] \tag{5.15}$$

$$|[\alpha, \beta]| = |[\overline{\beta}, \overline{\alpha}]| \tag{5.16}$$

Specific instances of these properties are used further on, namely saving interval sizes from $\bot$ only for equivalence classes and not for all MBFs

$$\forall \texttt{ permutation } p : |[\bot, \beta]| = |[\bot, p(\beta)]| \tag{5.17}$$

And getting all top interval sizes from the bottom interval sizes:

$$|[\alpha, \top]| = |[\bot, \overline{\alpha}]| \tag{5.18}$$

5.6 Interval Iteration

In the later chapters we want to iterate over all MBFs within an interval. This is useful for validating fast intervalSize algorithms, and for the earlier stages of the P-Coëfficient formula. To understand the generic interval iteration algorithm we first have to start with the simplest case and gradually make it more generic until we can iterate arbitrary intervals. In the end we are left with an elegant algorithm using a tiny $O(2^N)$ space, and $O(\binom{N}{\lfloor N/2 \rfloor}|[\alpha, \beta]|)$ time.

5.6.1 Full iteration: $[\bot, \top]$

This is the simplest iteration algorithm, we just have to iterate every single MBF in N variables *once*. A simple idea would be to just start from the empty MBF $\bot$, and then recursively keep adding bits one bit at a time until we reach the full MBF $\top$. That would work, except for the fact that most MBFs are reachable via more than one path, and hence duplicates will arise. For example $\{\} \to \{a\} \to \{a, b\}$, and $\{\} \to \{b\} \to \{a, b\}$ are both paths leading to $\{a, b\}$, and hence we would visit $\{a, b\}$ twice. We could of course get around this issue by building a big set of every MBF we've visited, and checking if we've seen a specific MBF before adding it. Though

that would require $O(|[\bot, \top]|)$ memory. We can do much better than this. The trick is to look at the ways we could have reached each MBF. The most recently added element must be unique for every 'source' MBF we could have come from. Listing all possible paths we could have taken allows us to just pick exactly one path we accept, continuing if we come from there, and discarding paths that don't. This is done on the line marked [*]. A visualization of this structure is shown in Figure 5.10.

Proof of no duplication by induction on uniqueness of smaller MBFs:
Base Case: $\bot$ is unique. We only visit it once at the start of the recursion.
Induction:
We wish to prove that a newly generated MBF α will be unique. We can make a list of all possible most recent extentions $\{x_i \mid !\exists y \in \alpha : y \neq x_i\}$, each of which will correspond to exactly one smaller MBF β_i such that $\beta_i, x_i = \alpha$. $|\beta_i| = |\alpha| - 1$ There are no duplicates in this set as each one is missing a different element. We can also assume that each of those smaller MBFs are only reached once as well by the induction assumption. If we only continue if we come from one arbitrary MBF in this set, then our larger MBF will also be reached only once. □

```
1  generates MBF forEachMBFRecurse(MBF cur) {
2    yield cur;
3    AC extentions = cur.succ() & ¬cur; // find all 1-bit extentions
4    for(e : extentions) { // for each possible extention
5      MBF extended = cur | e; // add the extention
6
7      // list the possible paths we could have arrived at extended from
8      AC possiblePaths = extended & ¬extended.pred(); // [*]
9
10     // only recurse over the extention if we come from an arbitrarily
11     // chosen path (in this case the first one in the list)
12     // this way we will only recurse on any given MBF once,
13     // removing the duplicates.
14     if(possiblePaths.getFirst() == e) {
15       forEachMBFRecurse(extended);
16     }
17   }
18 }
19 generates MBF forEachMBF() {
20   forEachMBFRecurse(⊥);
21 }
```

5.6.2 Iteration up to a specific MBF: $[\bot, \beta]$

This basic algorithm, that iterates from $\bot$ to $\top$, can be altered to iterate only up to a certain top MBF. Just filter the possible extentions on whether they are still below β: $\forall$ *extentione* : $e \leq \beta$. Due to the way monotonic boolean functions are structured, this comes down to a simple AND between the extentions and the top. This is done at the line marked [*].

```
generates MBF forEachMBFUpToRecurse(MBF cur, MBF top) {
  yield cur;
  // Find all extentions, but remove extentions
  // that would lead us to MBFs not below top
  AC extentions = cur.succ() & ¬cur & top; // [*]
  for(e : extentions) {
    MBF extended = cur | e;
    AC possiblePaths = extended & ¬extended.pred();
    if(possiblePaths.getFirst() == e) { // deduplicate
      forEachMBFUpToRecurse(extended, top);
    }
  }
}
generates MBF forEachMBFUpTo(MBF top) {
  forEachMBFUpToRecurse(⊥, top);
}
```

5.6.3 Iteration of arbitrary intervals: $[\alpha, \beta]$

We would like to just start the recursion at α instead of $\bot$. The problem with this is that in the deduplication, we assumed that all possible paths leading to an MBF would be followed. This was not a problem in the previous algorithm since the MBFs we excluded could never lead to an MBF we didn't exclude. But here, since we are excluding MBFs from the bottom out, many paths won't be reached. The solution is to modify the deduplication code to account for this, and only look at the paths that could have been reached from α. The line marked [*] is the extra filter. The reasoning is as follows: All incoming paths we wish to keep are paths p for which $\alpha <= p$. For MBFs this means that p is a superset of α. Any path where the newest added bit is a bit in α means that it came from an MBF which didn't have that bit, which means it wasn't a superset of α, hence it should be removed.

```
generates MBF forEachMBFBetweenRecurse(MBF cur, MBF bot, MBF top) {
  yield cur;
  // add all extentions under top
  AC extentions = cur.succ() & ¬cur & top;
  for(e : extentions) {
    MBF extended = cur | e;

    // instead of limiting the possible extentions as we did for top,
    // here we have to limit the set of paths that we could have
    // reached extended from, to account for paths that would have
    // been blocked by bot, so we don't remove the only path
    AC possiblePaths = extended & ¬extended.pred() & ¬bot; // [*]
    if(possiblePaths.getFirst() == e) { // deduplicate
      forEachMBFBetweenRecurse(extended, bot, top);
    }
  }
}
generates MBF forEachMBFBetween(MBF bot, MBF top) {
  forEachMBFBetweenRecurse(bot, bot, top);
}
```

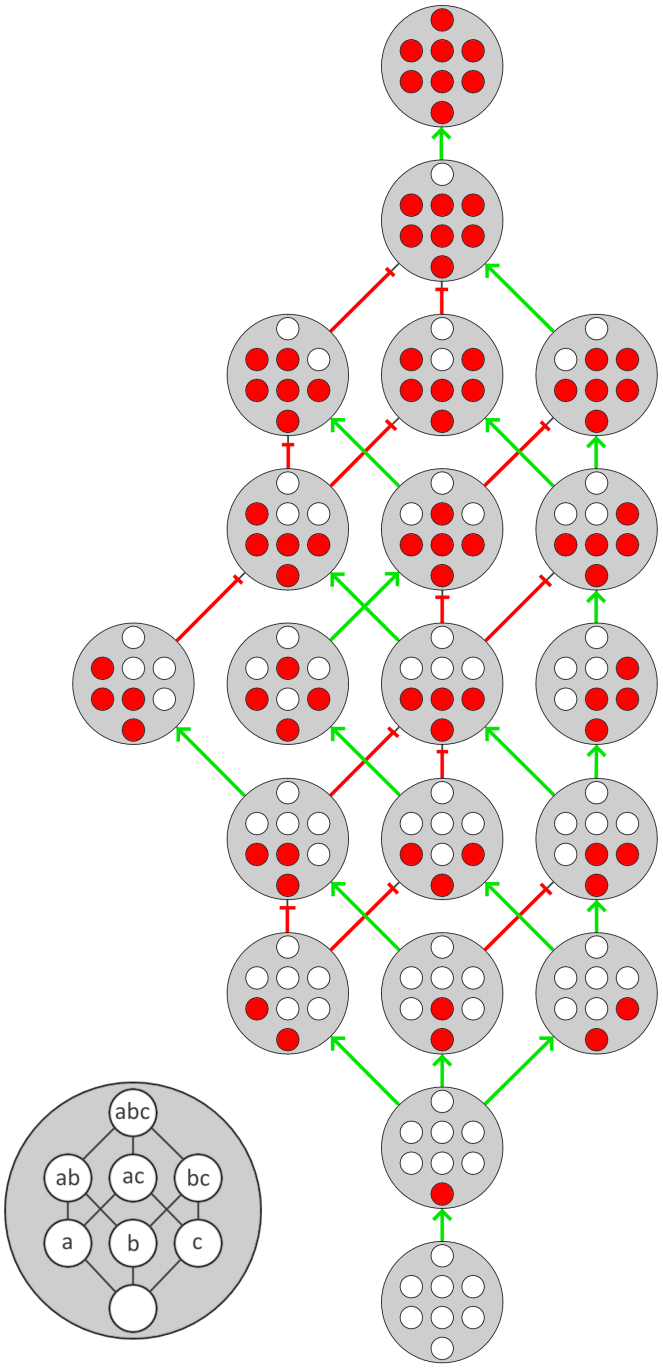

Figure 5.10: Structure of iterating the whole MBF space for 3 variables. The green arrows form a tree which visits every MBF exactly once. The red lines are blocked of paths because they were duplicates. Notice that for every MBF, exactly one path is let through, all others are blocked.

Chapter 6

Finding all Equivalence Classes for 7 Variables

The naive method for finding all Equivalence Classes is to iterate over the whole MBF space, canonizing every single one and creating groups based on that. For 7 variables, that gives us on average about 5000 canonizations per Equivalence Class. With this improved algorithm we can drastically reduce this, to about 16 canonizations per Equivalence Class. To do this, we must first bring some structure into the MBF space. We will first divide all MBFs into size classes, based on the number of *True* elements they have. So $\bot$ will be in size class 0, and $\top$ will be in size class 2^N, with all other MBFs somewhere inbetween. Like the layering structure seen in Figure 3.3. Evidently all MBFs in one Equivalence Class will be in the same size class. Therefore we can re-represent this structure in terms of the Equivalence Classes, as shown in Figure 6.1. We will compute all Equivalence Classes by following the structure on the right, from bottom to top, and we will do this size class by size class. For every Equivalence Class in the current size class S_i, find all MBFs that are one-element extentions of the representative of the current Equivalence Class. Canonize every MBF we find, and remove all the duplicates. Now we have all Equivalence Classes in size class S_{i+1}. Repeat for all size classes.

We must prove that every Equivalence Class in S_{i+1} is visited at least once by this method. We will prove this by induction over uniqueness of classes of size i:
Base case: The only equivalence class with size 0 is $\bot$. The iteration starts at $\bot$, so it is visited once.
Induction: Let us look at the contractions of a given class $\alpha = (\gamma, x) \in S_{i+1}$. (With γ the contracted MBF of size i, and x the contraction.) By contracting the class by one element, we get a new MBF $\gamma \in S_i$. This MBF is part of some equivalence class $\gamma \simeq \beta \in S_i$. Permuting γ by a permutation p such that $p(\gamma) = \beta$, and adding the permuted x: $p(x)$ gives us a permuted $p(\alpha)$, still part of the equivalence class of α: $\alpha \simeq p(\alpha) = (p(\gamma), p(x)) = (\beta, p(x))$. We have now proved that for every Equivalence Class represented by $\alpha \in S_{i+1}$ there exists a $\beta \in S_i$ that has an extention $p(x)$ that gives us an MBF in the class of α. Thus all Equivalence Classes will be found. $\square$

```
set<MBF> nextSizeClass(set<MBF> curSizeClass) {
  set<MBF> result;

  for(MBF eqClass : curSizeClass) {
    AC extentions = eqClass.succ() & ¬eqClass;
    for(e : extentions) { // for each possible extention:
      MBF extended = eqClass | e; // add it
      // discover extended's equivalence class by canonizing
      MBF extendedEqClass = extended.canonize();
      if(!result.contains(extendedEqClass)) {
        result.insert(extendedEqClass);
      }
    }
  }

  return result;
}
```

6.1 Parallelization and custom `set` Data Structure

This problem is not trivially parallelizeable, there is contention on the result set, which has to be protected with a mutex:

```
set<MBF> nextSizeClass(set<MBF> curSizeClass) {
  set<MBF> result;
  mutex resultMtx;

  // Iterate using multiple threads
  parallelfor(MBF eqClass : curSizeClass) {
    AC extentions = eqClass.succ() & ¬eqClass;
    for(e : extentions) {
      MBF extended = eqClass | e;
      MBF extendedEqClass = extended.canonize();

      resultMtx.lock(); // heavy contention on this critical section
      if(!result.contains(extendedEqClass)) {
        result.insert(extendedEqClass);
      }
      resultMtx.unlock();
    }
  }
  return result;
}
```

The problem here is that the cure is worse than the disease. Canonisations are expensive, but the mutex locking and unlocking is even more expensive. A solution for this would be to have a mutex-less `try_contains` first, which will return false with certainty if it does not contain the object, and tries to return true if it does, but is allowed to return false anyways. We can use this as an early-exit so we don't have to take a full mutex lock if we don't need it. Of course, this contains method does not give certainty, so the real contains must still be checked while the lock is held.

```
set<MBF> nextSizeClass(set<MBF> curSizeClass) {
  set<MBF> result;
  mutex resultMtx;

  parallelfor(MBF eqClass : curSizeClass) {
    AC extentions = eqClass.succ() & ¬eqClass;
    for(e : extentions) {
      MBF extended = eqClass | e;
      MBF extendedEqClass = extended.canonize();

      // This is outside of resultMtx' critical section.
      // We're allowed to do this because try_contains can run
      // concurrently with insert, due to its atomic implementation
      if(result.try_contains(extendedEqClass)) {
        continue; // already found, early exit
      }

      resultMtx.lock(); // less contention due to early exits
      if(!result.contains(extendedEqClass)) {
        result.insert(extendedEqClass);
      }
      resultMtx.unlock();
    }
  }

  return result;
}
```

While this is a nice solution, The Standard Library's `set` does not provide a thread-safe `try_contains`. That's because `std::set` is only thread-compatible. The `result.try_contains(...)` is not protected by a `resultMtx`, so it can be called at the same time as `result.insert(...)`. The reason `std::set` can't provide such a method is because it's a dynamic container. The underlying data buffer may be reallocated any time an insert is done to create more space. If at the same time a `try_contains` was inspecting the buffer, then we get a memory protection error or undefined behaviour. In fact, if we wish to implement `try_contains` for our own set type, then we can't dynamically reallocate our buffer. The buffer must be a fixed size after creation. We also can't allow removal of elements, as in that case `try_contains` can't give us any information at all. Because of these constraints, and if we are very careful, then it is possible to build a `BufferedSet` using atomic hash table entries which has a lock-free thread-safe `try_contains` method. Though `insert` still requires synchronization!

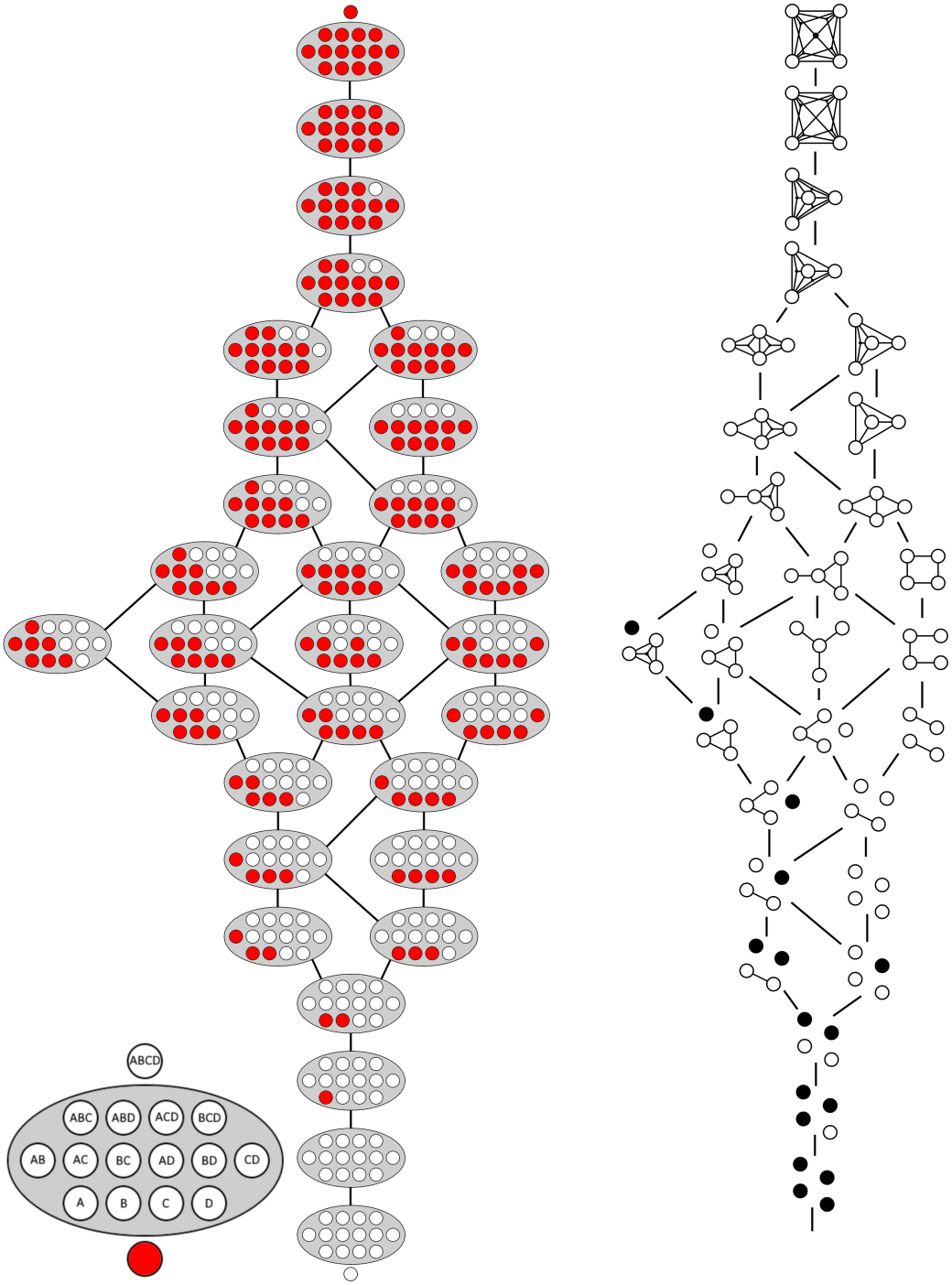

Figure 6.1: All equivalence classes in the lattice of 4 Variables. Shown as hypergraphs (right), and a chosen representative (left). Left: for compactness the $\emptyset$ and $\{abcd\}$ nodes are omitted as they always have the same value, except for $\top$ and $\bot$. Right: Singleton hyperedges are represented as empty nodes, hyperedges of arity 3-4 connect 3 nodes with a dot.

```
template<typename T>
class BufferedSet {
  class SetNode {
    T object;
    SetNode* nextNode;
  };

  SetNode* data; // fixed after creation
  size_t capacity;
  atomic<SetNode*>* hashTable; // fixed after creation
  size_t bucketCount;
  atomic<size_t> size = 0;

public:
  BufferedSet(size_t capacity, size_t bucketCount) :
    data(new SetNode[capacity]),
    capacity(capacity),
    hashTable(new atomic<SetNode*>[bucketCount](nullptr)),
    bucketCount(bucketCount) {}
  ~BufferedSet() {/*deletes...*/}

  void insert(T item) {// does not check contains
    SetNode& claimedObject = this->data[this->size.fetch_add(1)];
    atomic<SetNode*>& bucket = item.hash() % this->bucketCount;
    claimedObject.object = item;
    claimedObject.nextNode = bucket.load();
    bucket.store(&claimedObject);
  }
  bool try_contains(T item) const {
    atomic<SetNode*>& bucket = item.hash() % this->bucketCount;
    SetNode* curNode = bucket.load();
    // New insertions after this line is executed won't be found.
    // This is where the try_ comes in, it may return false
    // even if the object was added after this line.
    while(curNode != nullptr) {
      if(curNode->object == item) {
        return true;
      }
      curNode = curNode->nextNode;
    }
    return false;
  }
  // just try_contains requiring lock to be held before calling
  bool contains(T item) const {return try_contains(item);}
};
```

Using this data structure, efficient implementation of `nextSizeClass` is possible. Of course, the real `BufferedSet` implementation is more complicated, mostly combining the `if(!contains(...)){insert(...)}` into one method as an optimization. After adding all new Equivalence Classes to the `BufferedSet`, we can optimize this data structure to increase spatial locality and decreased memory usage through the process of 'baking'. This is done by sorting all values by bucket, so elements in the same bucket appear consecutively (improving cache locality). This also allows us to get

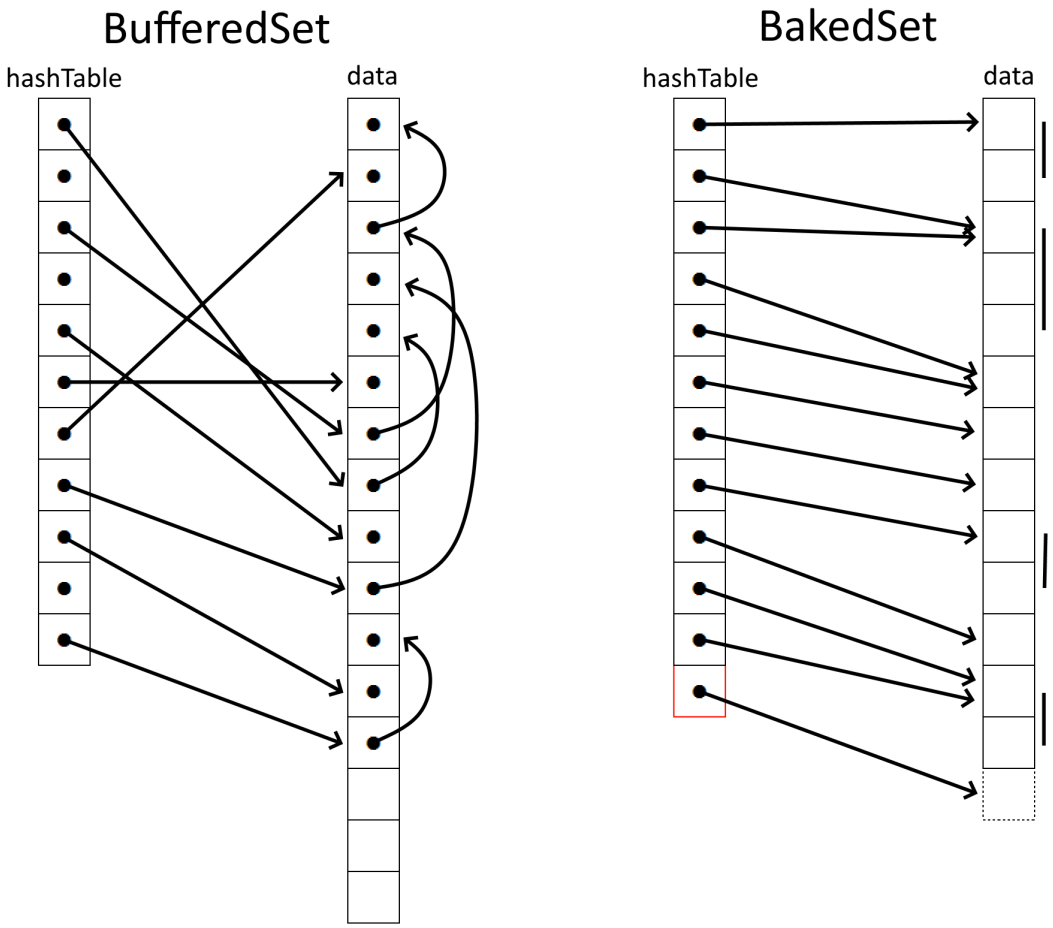

Figure 6.2: BufferedSet vs BakedSet

rid of the `nextNode` pointer, saving 4GB of space for D(7). To iterate over a bucket, we iterate from the node pointed to by the current bucket, up to the node pointed to by the next bucket. This means that we encode the number of elements of the bucket into the difference between this bucket and the next bucket pointer. We must therefore include an extra bucket at the end that points to the end of the data section.

Chapter 7

Precomputed values Data Structure

Now that we have all Equivalence Classes, we can build a data structure for storing pre-computed information. In order to store this extra information, we need to convert our `BakedSet`s to `BakedMap`s. This is trivial to do, just copying over all MBFs, adding some extra space after every MBF for the extra data, and modifying the hashTable accordingly. One piece of such precomputed information is the Equivalence Class Size, meaning the number of distinct MBFs that are in the group. We can compute this by iterating over all $n!$ permutations of the class representative, and counting the number of duplicates of the representative. Every permutation will have the same number of duplicates, so the size of the equivalence class with representative α is

$$D_\alpha = \frac{n!}{Duplicates_\alpha} = \frac{n!}{\sum_{\substack{\gamma \in Permut_\alpha \\ \gamma = \alpha}} 1} \tag{7.1}$$

One could ask why we do this instead of iterating over all MBFs, since we'll end up performing more than $D(7)$ iterations anyway. The reason is that permutations are much cheaper than canonizations. One swap takes on the order of 2ns whereas a canonization takes ∼380ns. By starting from the equivalence classes we can avoid having to canonize every single MBF. Computing this for all Equivalence Classes $R(7)$ takes 1.5 hours on my 4-core i5.
Another datapoint we wish to add are all interval sizes $|[\bot, \alpha]|$, which are used in the P-Coëfficient method [1]. The computation of these interval sizes is described in Section 8.
Finally, we wish to bring some structure into the data, to iterate more efficiently over 'child' equivalence classes, that is, all equivalence classes that have a permutation that is a subset of the starting class. We'll do this by mirroring the structure shown in Figure 6.1. We could have generated the links when we were generating the classes in the first place, but that would add more complexity and synchronization overhead

[1] See Section 9

to an already difficult algorithm. So we generate the links separately. We can do this the exact same way we found all the equivalence classes,

7.1 Considerations for large data structure for 7 Variables

To start, we wish to minimize the size of the extra data we need to store, so we start out by figuring out exactly how many bits we need for every piece. For 7 Variables, the largest interval size we can get is $|[\bot_7, \top_7]| = D(7) = 2414682040998 \simeq 2^{41.13}$, so we need at least 42 bits. The largest group size possible is $n! = 7! = 5040 \simeq 2^{12.29}$, so we need at least 13 bits. For ease, we can group the interval size, and equivalence class size together into a 64-bit block, allocating 48 bits for the interval size, and 16 bits for the equivalence class size. We'll need another 64 bits for a pointer to the list of downlinks. One such element is shown in Figure 7.1

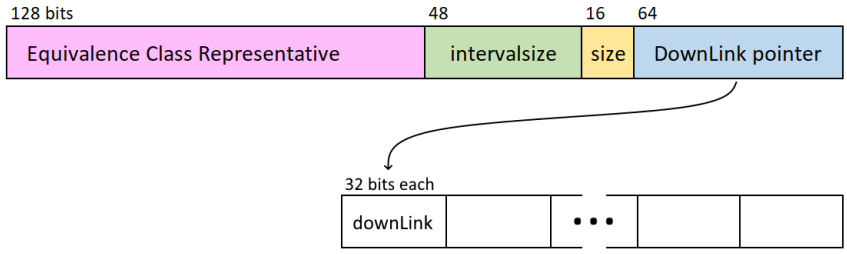

Figure 7.1: Structure of one map element. Bit widths shown.

Looking at how we store the downlinks, for every Equivalence Class there are about 10-20 downlinks. Adding them up gives us about $7 * 10^9$ links in total, so we have to think carefully about storage. The number of links per class is also quite small, so having all these separate lists adds a lot of allocation overhead[2] and cache inefficiency. Another memory improvement is to use 32-bit indices into the `BakedMap` data buffer instead of 64-bit pointers for the downlinks, since there are only 490'000'000 Equivalence Classes, which is slightly less than the ~2'000'000'000 32-bit integer limit. Because we now store all the links consecutively into one massive buffer, we can use the same trick like we used for `BakedSet` to store the size of the lists in the difference between the current, and next pointer, as demonstrated in Figure 7.2. Both of these considerations save us a massive amount of memory, considering that the final memory footprint of just the downLink buffer is 29GB, down from ~75GB had we not done these improvements.
The total memory footprint of the whole data structure is then about 45GB. You may ask yourself why we went through all this trouble to reduce this memory consumption, why we don't just use compute nodes with 100GB, or 200GB of memory. It's true,

[2]Every allocation takes with it a preamble listing the allocation size, and a node in the allocation table. This leads to an extra memory overhead of 16-32 bytes, for a small list of maybe 11 32-bit integers, that's 44 bytes of data, and 16-32 bytes of overhead. Adding to that alignment requirements, that may introduce yet more padding bytes.

the compute nodes we use have 192GB, or 256GB of memory, so the unoptimized data structure would still fit just fine. The reason we do this isn't to save on memory usage, but memory bandwidth. This data structure is so large that it couldn't possibly fit in cache[3], which means that pretty much all accesses will have to be pulled from memory. If the processing required per MBF is very small (as is the case in Section 10) then the limited throughput of memory accesses will become the bottleneck.

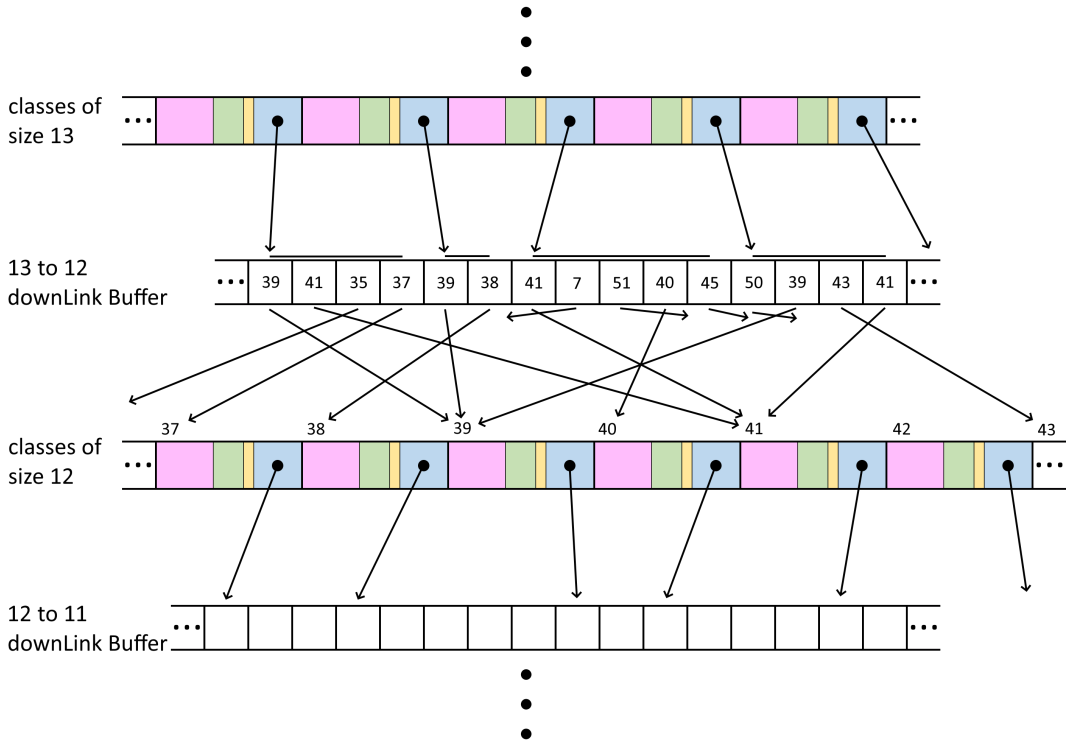

Figure 7.2: Structure the data buffers of each Size Class, and the downLink buffers connecting them.

7.2 Iterating child equivalence classes

We wish to iterate over all equivalence classes that have at least one MBF that is smaller than the representative of an equivalence class α. We can do this layer-by-layer by following the downLinks we built previously. If we just follow the starting class' downlinks, and then follow their downlinks and so on, we'll do this, but we'll also have a lot of duplicates. To deduplicate, we have to somehow keep track of which elements we've visited. We can do this by storing a 'visited' boolean for every equivalence class, setting it to one if we pass it by downlink. That way we know

[3]Typically around 20MB, though can go as high as 250MB for high-end server cpus

when we come across the same equivalence class twice. Naturally we cannot store this information into the data structure directly, as many threads will be reading (and then also writing) at the same time. Every thread must get their own 'visited' buffer. We can index into this buffer using the indices from the main data structure. Further improvements can be made by processing multiple equivalence classes at once, and storing bitsets into this visited buffer instead of individual bits, which can help to reduce memory bandwidth by reusing the same fetches for multiple equivalence classes.

Chapter 8

Intervals of D(7)

The interval sizes from $\bot$ to all MBFs (or equivalently all MBFs to top) are needed to use the P-coefficient method described in [6].

8.1 Naive interval size computation from $\bot$

$$|[\bot, \alpha]| = \sum_{\substack{\beta \in A_n \\ \beta \leq \alpha}} 1 \tag{8.1}$$

The naive method to do this, is just iterating over all MBFs that are smaller than the given MBF, using the algorithm denoted in Section 5.6.2. The problem is that the amount of work rises incredibly quickly, both in work per interval, as in the number of intervals, making this method infeasible for the intervals of $D(7)$. More efficient algorithms are therefore needed.

8.2 Incremental interval size computation

This new method reuses the previous results of smaller intervals, to build the interval sizes for larger ones. It is useless for computing individual interval sizes, but works wonders for computing all interval sizes at once, which is exactly what we need for the P-coëfficient method. For this method we only need interval sizes from $\bot$, but this method can compute all interval sizes from an arbitrary lower bound.
The main way the incremental part works is as shown in Figures 8.1, 8.2, and 8.3. Let's say we wish to compute the interval size of the interval $|[\bot, \alpha]|$ as shown in figure 8.1. Assuming we already have the interval sizes for all MBFs of size $|\alpha| - 1$, then we can pick one to build off of, let's say we have the interval size for the monotonic β as shown in Figure 8.2: $|[\bot, \beta]|$. The difference is the added node *bce*. The extra MBFs that are included in $[\bot, \alpha]$ must contain this new node, since otherwise it would have been in $[\bot, \beta]$. Expressed mathemathically using AntiChains:

$$\begin{aligned} &\forall \gamma \in A_n, \beta \in A_n, x \in succ(\beta) : \\ &\gamma \in [\bot, (\beta, x)] \implies \gamma \in [\bot, \beta] \texttt{ XOR } x \in \gamma \end{aligned} \tag{8.2}$$

This gives us

$$|[\bot, (\beta, x)]| = |[\bot, \beta]| + \sum_{\substack{\gamma \in [\bot, (\beta, x)] \\ x \in \gamma}} 1 \tag{8.3}$$

Since we have $|[\bot, \beta]|$ all that is left is to count the extra MBFs. Have a look at figure 8.3, the requirement that *bce* be enabled saves us a lot of work, since all elements below *bce* are also forced on. The left over choices (marked in blue) must still be counted exhaustively. This is also the reason why choosing the added element of the highest possible layer is ideal for performance, as that blocks out as many choices as possible. An example of such a graph with left over choices is Figure 8.4.

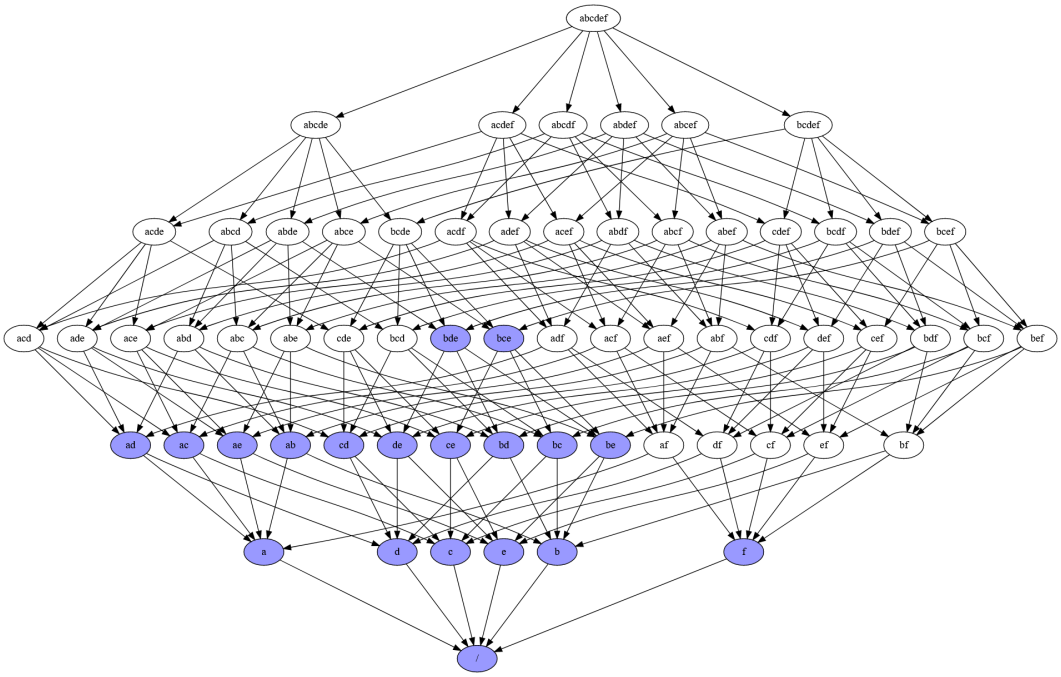

Figure 8.1: Interval for which we wish to compute the size. The MBF shown here is the upper bound of the interval, the lower bound is $\bot$, the empty MBF. All following interval graphs follow this convention.

```
1  int extendedIntervalSize(map<MBF, int> prevSizes, MBF cur) {
2    Layer topLayer = cur.getTopLayer(); // pick as high as possible
3    BF chosenElement = topLayer.first(); // pick an arbitrary element
4    MBF smallerMBF = cur & ¬chosenElement; // remove chosenElement
5    int smallerMBFIntervalSize = prevSizes[smallerMBF];
6    // elements below chosenElement can no longer be chosen
7    MBF blockedElements = chosenElement.monotonizeDown();
8    BF leftOverChoices = cur & ¬blockedElements;
9    return smallerMBFIntervalSize + countChoices(leftOverChoices);
10 }
```

A possible choice is defined as an assignment of *True* and *False* values to the elements with the constraint that internal monotonicity between elements is preserved. It is

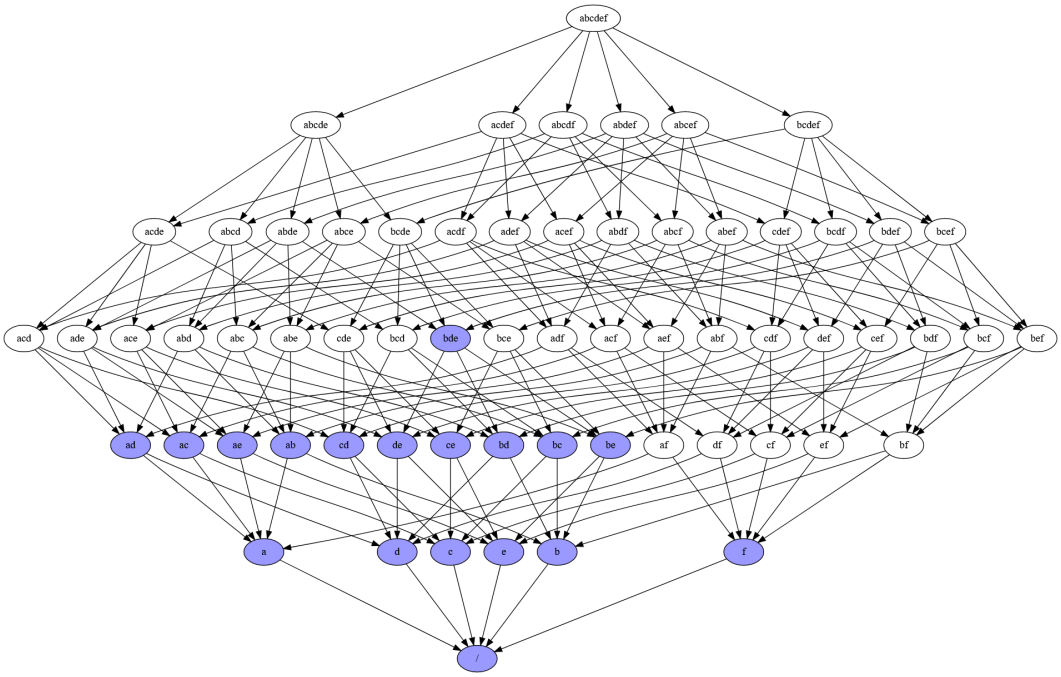

Figure 8.2: Smaller interval with known size

assumed that all elements outside of the chooseable set have already been fixed, and they add no additional constraints to the chooseable set.
A basic recursive countChoices implementation would be picking an arbitrary element, choosing it *True* or *False*, removing the now forced elements, and adding up the remaining number of choices for the left over elements.

```
int countChoices(BF choices) {
  if(|choices| == 0) {
    return 1;
  } else {
    BF elementToChoose = choices.first();

    // choose the element as False
    // force all elements above it as False
    BF newChoicesFalse =
      choices & ¬elementToChoose.monotonizeUp();

    // choose the element as True
    // force all elements above it as True
    BF newChoicesTrue =
      choices & ¬elementToChoose.monotonizeDown();

    return countChoices(newChoicesFalse)
       + countChoices(newChoicesTrue);
  }
}
```

To reduce the number of *monotonize* calls and the recursion depth, we can modify this

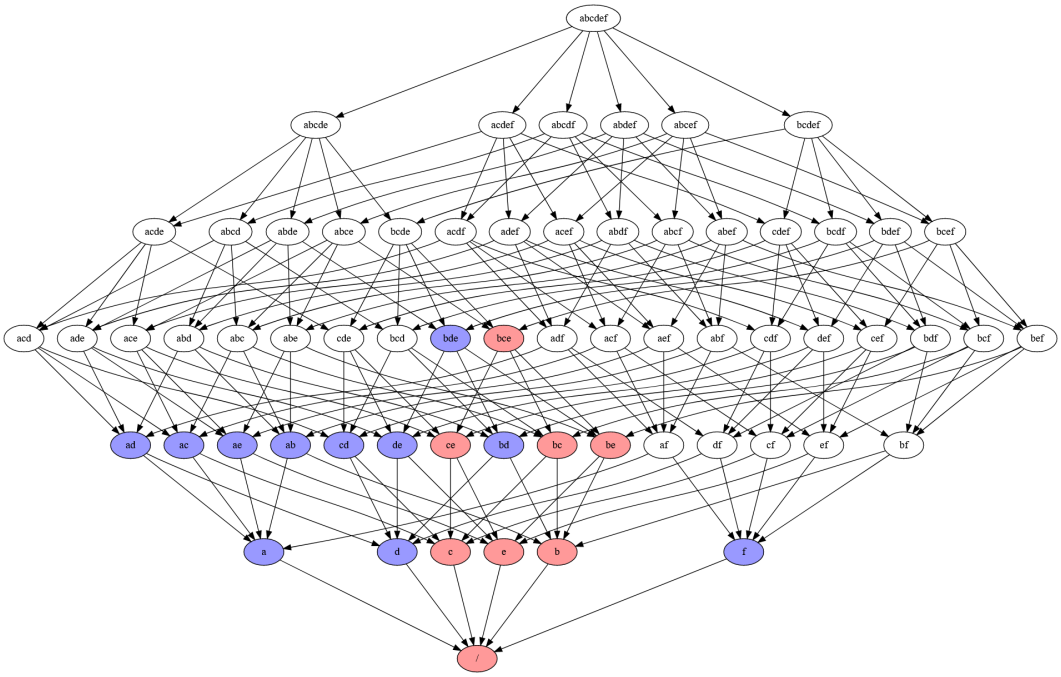

Figure 8.3: Nodes blocked by adding BCE

algorithm to work layer-by layer. Since within a layer all elements are independent, they can be chosen at once, needing only one *monotonizeUp* call per configuration. This gives us the additional benefit of being able to discard all *monotonizeDown* calls, since we'll always be working on the bottom layer of the enabled elements.

```
1  int countChoices(BF choices) {
2    if(|choices| == 0) {
3      return 1;
4    } else {
5      // returns the lowest layer of the BF that has enabled elements.
6      Layer chosenHere = choices.getBottomLayer();
7
8      int totalChoices = 0;
9      for(Layer chosenFalse : chosenHere.powerset()) {
10       BF forcedFalse = chosenFalse.monotonizeUp();
11       BF forcedElements = chosenHere | forcedFalse;
12       totalChoices += countChoices(choices & ¬forcedElements);
13     }
14
15     return totalChoices;
16   }
17 }
```

We can still improve this further, once we arrive at the final top layer, in all of the leaf recursions, all elements are independent. Instead of still iterating over the full powerset, we can just return the size of the powerset $2^{|\texttt{choices}|}$.

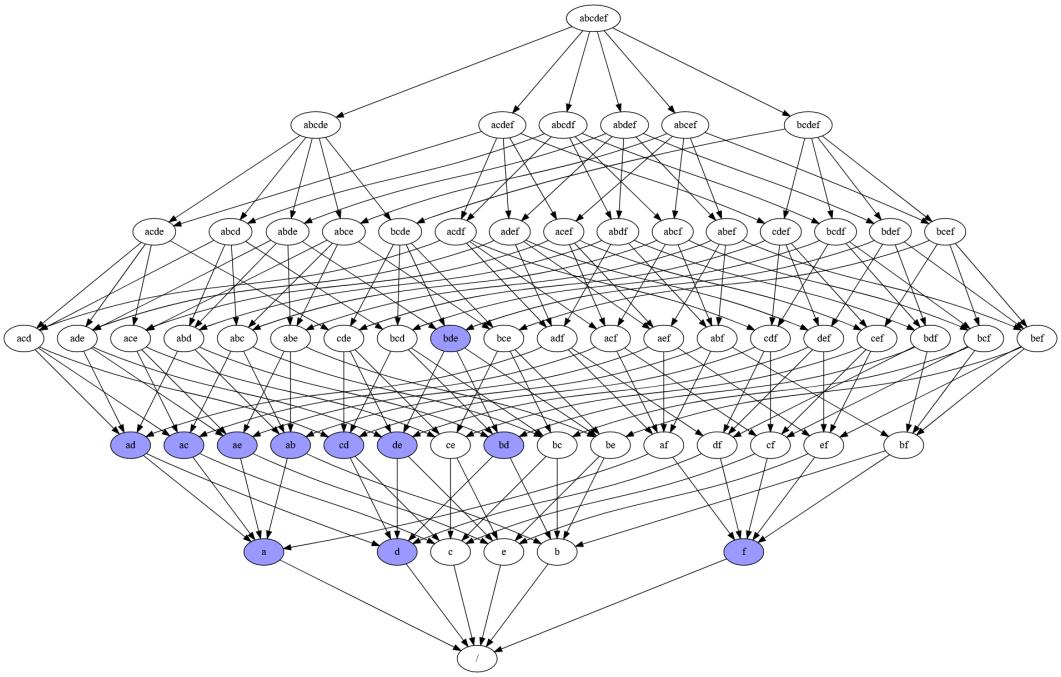

Figure 8.4: Leftover nodes to count choices for

```
int countChoices(BF choices) {
  // returns the lowest layer of the BF that has enabled elements.
  Layer chosenHere = choices.getBottomLayer();
  if(choices == chosenHere) {
    return 2^|choices|;
  } else {
    int totalChoices = 0;
    for(Layer chosenFalse : chosenHere.powerset()) {
      BF forcedFalse = chosenFalse.monotonizeUp();
      BF forcedElements = chosenHere | forcedFalse;
      totalChoices += countChoices(choices & ¬forcedElements);
    }

    return totalChoices;
  }
}
```

A similar trick, named 'Layer Skipping' is used in [6], Theorem 6.2.

While this already drastically improves the performance, it is still not ideal. For one, it's completely defeated by a 1-element top layer. We need to make larger powers of 2. Observe that the only requirement for using the powers is that the elements are independent, they do not necessarily have to be in the same Layer. A good approximation for this very wide set of independent elements is all elements which do not have an element above them, kind of like the top 'surface' of the choices set, as shown in Figure 8.5.

```
int countChoicesRecursive(BF choices, AC independentElements) {
  BF dependent = choices & ¬independentElements;

  if(|dependent| == 0) { // only independent elements left
    return 2^|choices|;
  } else {
    // only choose non-independent elements
    Layer chosenHere = dependent.getBottomLayer();
    int totalChoices = 0;
    for(Layer chosenFalse : chosenHere.powerset()) {
      BF forcedFalse = chosenFalse.monotonizeUp();
      BF forcedElements = chosenHere | forcedFalse;
      totalChoices += countChoicesRecursive(
        dependent & ¬forcedElements, independentElements
      );
    }

    return totalChoices;
  }
}
int countChoices(BF choices) {
  // This monotonizeDown().asAntiChain() operation removes all
  // elements that have another element above them for general BFs
  AC independentElements = choices.monotonizeDown().asAntiChain();
  return countChoicesRecursive(choices, independentElements);
}
```

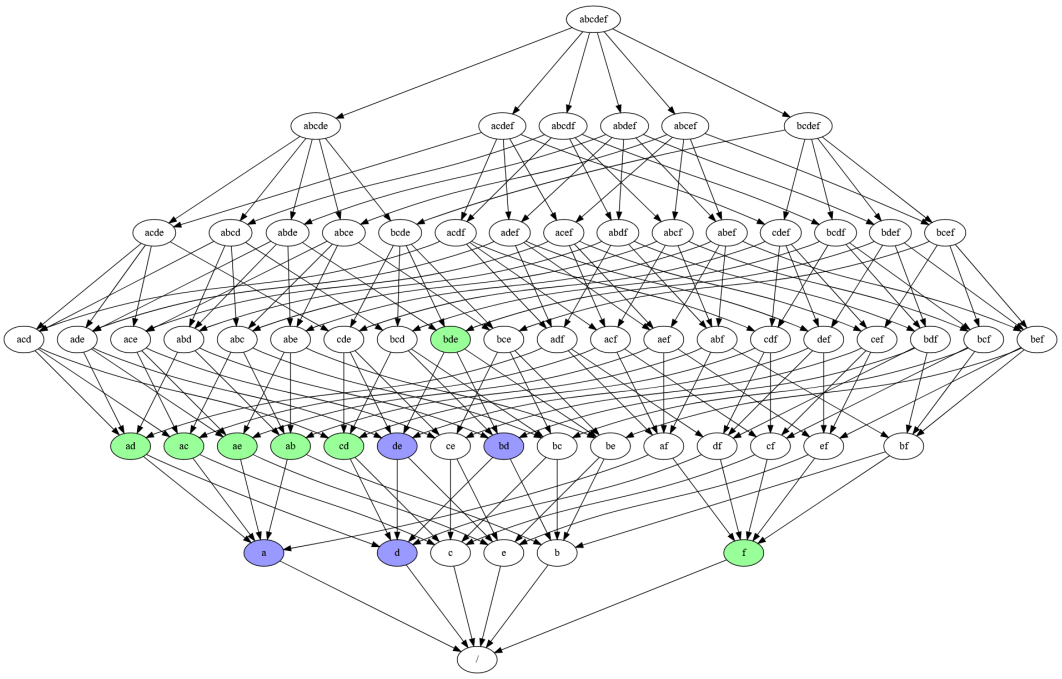

Figure 8.5: Subsets of Nonconflicting elements can be counted with powers of 2

Finally, a small optimisation can still be done, to make this widest layer approximation

even wider, and that is for structures with a small (1-4 elements) top layer, removing the top layer. This could reveal a few more elements in the layer below, and enlarge the independent set, as shown comparing Figures 8.5, 8.6. Though of course, we should only do this if in fact it increases the size of the independent set. But if we can make an improvement by removing the top layer, we have to take into account the choices that top layer provided.

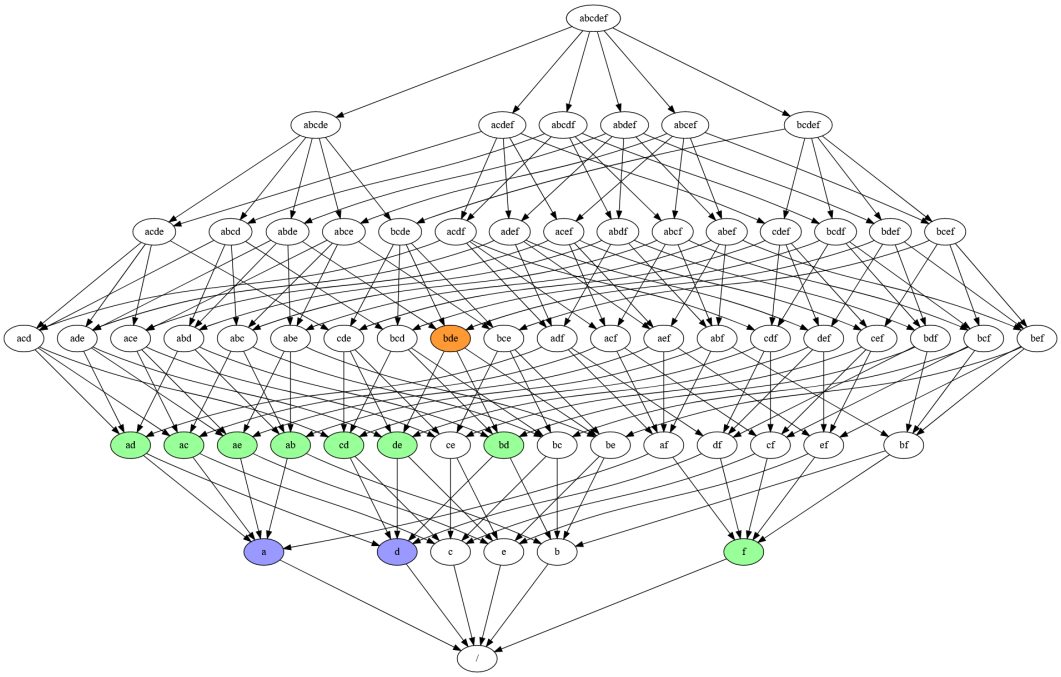

Figure 8.6: Widest AntiChain, to maximize powers of 2

```
int countChoices(BF choices) {
  AC independentElements = choices.monotonizeDown().asAntiChain();

  Layer topLayer = choices.getTopLayer();
  BF choicesWithoutTop = choices & ¬topLayer;
  AC improvedIndep = choicesWithoutTop.monotonizeDown().asAntiChain();
  if(|improvedIndep| > |independentElements|) {// removing top helps!
    int totalChoices = 0;

    for(Layer chosenTrue : topLayer.powerset()) {
      BF forcedTrue = chosenTrue.monotonizeDown();
      BF choicesLeft = choicesWithoutTop & ¬forcedTrue;
      totalChoices += countChoicesRecursive(choices, improvedIndep);
    }
    return totalChoices;
  } else {
    return countChoicesRecursive(choices, independentElements);
  }
}
```

8.3 AVX and AVX-512 optimisations

Now that the algorithm is finalized, it is time to really optimize towards the CPU architecture that will be running our code. Most modern Intel and AMD cpus support AVX and AVX2, and Intel CPUs for HPC often also support AVX-512.[22] These special SIMD instruction set extentions allow the programmer to use instruction-level parallelism for higher throughput computing. If the problem is well suited for it, and the implementation is done correctly, these can provide up to 4x and 8x improvements for 64-bit operations.

Below is some sample AVX-512 code for performing 8 `monotonizeUp<6>` operations at the same time. This kind of low-level optimization can give tremendous speed boosts, but makes code more difficult to read, and maintain.

```
__m512i monotonizeUp6AVX512(__m512i bs) {
  // a,b, and c have to be masked out before shifting,
  // so we don't add an a to an element that already has an a
  __m512i v0Added = _mm512_slli_epi64(_mm512_and_si512(
            _mm512_set1_epi8(0b01010101), bs), 1);
  bs = _mm512_or_si512(bs, v0Added);
  __m512i v1Added = _mm512_slli_epi64(_mm512_and_si512(
            _mm512_set1_epi8(0b00110011), bs), 2);
  bs = _mm512_or_si512(bs, v1Added);
  __m512i v2Added = _mm512_slli_epi64(_mm512_and_si512(
            _mm512_set1_epi8(0b00001111), bs), 4);

  // for d,e, and f we can use a trick of using smaller shift blocks,
  // shifting out the duplicate variables. For example shifting by 8
  // in 16-bit blocks, that shifts out the 8 bits that already
  // contained d. This saves quite a lot of instructions and masks!
  bs = _mm512_or_si512(bs, v2Added);
  __m512i v3Added = _mm512_slli_epi16(bs, 8);
  bs = _mm512_or_si512(bs, v3Added);
  __m512i v4Added = _mm512_slli_epi32(bs, 16);
  bs = _mm512_or_si512(bs, v4Added);
  __m512i v5Added = _mm512_slli_epi64(bs, 32);
  bs = _mm512_or_si512(bs, v5Added);
  return bs;
}
```

8.4 Interval Sizes Performance

Performance measurements for the various iterations of the incremental algorithm, as well as individual algorithms are shown in Figure 8.7. Exponentials that suddenly stop were processes that had to be stopped since they wouldn't deliver a result in a reasonable amount of time. Only two measurements made it all the way to the end, being the AVX optimized version running on my laptop taking 19 hours, and the AVX-512 optimized version running on one Genius Skylake Node[23], taking 6:20 hours. Both returned the same set of intervals.

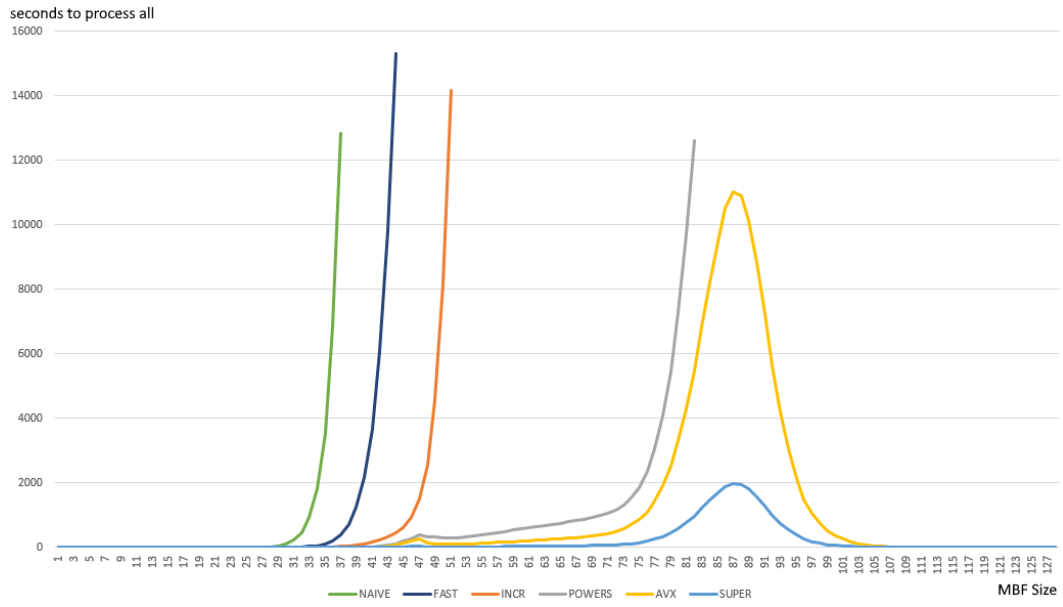

Figure 8.7: Performance measurements of the various algorithms tested for computing all intervals from $\bot$. All measurements were performed on an Intel i7-8850H 6-core laptop CPU, with 32GB of RAM, except the SUPER run, which was run on one node of the Genuis SuperCluster at VSC. (Dual Xeon Gold 6140 36-cores)

8.5 Verifying the interval sizes by computing $D(8)$

One major issue, that has to be contended with, is Cosmic Radiation. [24] Cosmic Rays can occasionally cause bitflips in memory. This is mostly mitigated by ECC memory, although it still happens. The more memory you use, and the longer you run your application, the more chance there is for an error. Since we are running high memory use computations for a long time, this risk exists. We must somehow be able to verify that the values we get are in fact correct. (Assuming we haven't made any logical errors in the code.) For the interval sizes, we can do a simple check by computing $D(8)$: This comes from

$$D(n+1) = \sum_{\gamma \in A_{(n+1)}} 1 \tag{8.4}$$

Any AntiChain/MBF can be split into two parts along a variable: the elements that contain the variable, and the elements that don't. Noted as $\gamma = \alpha \vee (\beta \times \{v\})$. One important constraint between α and β is that $\beta \leq \alpha$, this is to preserve monotonicity, we can thus write this sum as:

$$D(n+1) = \sum_{\alpha \in A_n} \sum_{\substack{\beta \in A_n \\ \beta \leq \alpha}} 1 \tag{8.5}$$

Seeing as the sum over betas is the definition for interval size from bot, we can rewrite it as follows:

$$D(n+1) = \sum_{\alpha \in A_n} |[\bot, \alpha]| \tag{8.6}$$

Finally, using the property that interval sizes from $\bot$ to α are invariant to permutations of α we finally arrive at the following:

$$D(n+1) = \sum_{\alpha \in R_n} D_\alpha |[\bot, \alpha]| \tag{8.7}$$

With D_α the number of MBFs in the Equivalence Class of α.
Computing this for our dataset of intervals gives us $D(8) = 56130437228687557907788$, which is the correct value, giving us more confidence that the intervals are indeed correct.

Chapter 9

Computing D(9) using P-coefficients

In their 2014 paper "On the number of antichains of sets in a finite universe"[6], Patrick De Causmaecker and Stefan De Wannemacker describe a general 'jumping' method for computing larger Dedekind numbers from the lattice structure with fewer variables. This allowed them to compute the 8th Dedekind number from the lattice structure of 6.
The P-coefficient formula:

$$D(n+k) = \sum_{\substack{\alpha,\beta \in A_n \\ \alpha \leq \beta}} |[\bot, \alpha]| P_{n,k,\alpha,\beta} |[\beta, \top]| \tag{9.1}$$

While it would be nice to pick $k = 3$ and just compute D(9) from the structure of 6, at this time no efficient methods exist for computing $P_{n,k,\alpha,\beta}$ for $k \geq 3$. So we have to use $k = 2$:

$$D(n+2) = \sum_{\substack{\alpha,\beta \in A_n \\ \alpha \leq \beta}} |[\bot, \alpha]| P_{n,2,\alpha,\beta} |[\beta, \top]| \tag{9.2}$$

Naively implementing this sum would result in having to compute $D(8) \approx 5.613 * 10^{22}$ terms.[1] With current hardware this is infeasible. Luckily we can improve this by several orders of magnitude.
We can split this sum into a sum over alphas, and a nested one over betas:

$$D(n+2) = \sum_{\alpha \in A_n} |[\bot, \alpha]| \sum_{\substack{\beta \in A_n \\ \alpha \leq \beta}} P_{n,2,\alpha,\beta} |[\beta, \top]| \tag{9.3}$$

We can now exploit the symmetries of α. We are only allowed to do this because:

1. $|[\bot, \alpha]|$ is invariant over permutation of α

[1] This is the same summation structure like we used in Section 8.5

2. $P_{n,2,\alpha,\beta}$ is invariant over applying the same permutation to both α and β. This still leaves to prove that any β for the original α will map to a new permuted β'. Luckily, the condition $\alpha \leq \beta$ is also invariant over applying the same permutation to both α and β, so this is proven.

$$D(n+2) = \sum_{\alpha \in R_n} D_\alpha |[\bot, \alpha]| \sum_{\substack{\beta \in A_n \\ \alpha \leq \beta}} P_{n,2,\alpha,\beta} |[\beta, \top]| \tag{9.4}$$

with R_n the set of equivalence class representatives, and D_α the size of the equivalence class α.

Removing all other values and taking just the sum structure, we can estimate the number of terms we would need. Much like in Section 8.5 we can replace the inner sum by $|[\alpha, \top]|$.

$$NumTerms_n = \sum_{\alpha \in R_n} \sum_{\substack{\beta \in A_n \\ \alpha \leq \beta}} 1 = \sum_{\alpha \in R_n} |[\alpha, \top]| \tag{9.5}$$

Computing this gives us $NumTerms_7 = 11483553838459169213 \approx 1.148 * 10^{19}$ terms, which is still an astronomical amount, but it enters the range of being computable on a supercomputer. Sadly whereas P-coefficients are relatively fast to compute, canonizations are not (Table 5.2). And we need these canonizations to do the $|[\beta, \top]|$ interval size lookups.[2] We can alleviate the need for canonization by iterating over the equivalence classes themselves - thereby getting the interval size for free - and just summing p coefficients for all MBFs in the equivalence class. Betas can be chosen efficiently by using the downlinks of the current equivalence classes. [3]

$$D(n+2) = \sum_{\alpha \in R_n} D_\alpha |[\bot, \alpha]| \sum_{\substack{\beta \in R_n \\ \exists \delta \simeq \beta : \alpha \leq \delta}} |[\beta, \top]| \sum_{\substack{\gamma \simeq \beta \\ \alpha \leq \gamma}} P_{n,2,\alpha,\gamma} \tag{9.6}$$

Again, we can't store all the elements in each equivalence class, we'd need $O(D(7))$ space, which is infeasible. Constructing all elements in the equivalence classes on the fly is also not terribly efficient, so I concluded that it's acceptable to just sum over all permutations, and divide out the duplicates afterwards. The number of duplicate permutations per class element is $\frac{n!}{D_\beta}$.

$$D(n+2) = \sum_{\alpha \in R_n} |[\bot, \alpha]| D_\alpha \sum_{\substack{\beta \in R_n \\ \exists \delta \simeq \beta : \alpha \leq \delta}} |[\beta, \top]| \frac{D_\beta}{n!} \sum_{\substack{\gamma \in Permut_\beta \\ \alpha \leq \gamma}} P_{n,2,\alpha,\gamma} \tag{9.7}$$

It turns out that this is not a huge overhead, as for the vast majority of equivalence classes $D_\beta = n!$. The average equivalence class size is $\frac{D(n)}{R(n)}$, which for $n = 7$ yields

[2] See Section 7

[3] See Section 7.2

4927.79, which compared to the number of permutations $7! = 5040$ gives us a waste of 2.22% which is an acceptable trade-off for a considerable simplification of the code. Another trade-off we made here was trading canonizations for $\leq$ checks. Since for the iteration of betas in Equation 9.4, we could use MBF iteration up to α from Section 5.6.2. However, once we start iterating over Equivalence Classes this trick no longer applies, and we have to check $\gamma \leq \alpha$ for every P-coefficient.
Measuring the actual rate at which this check passes gives us some staggering results, shown in Table 9.1. The measurement for 7 variables is not as accurate, since I had to use a small subset[4] of the data, but for everything ≤ 6 the full iteration was performed.

4	79.1495%
5	42.7662%
6	18.8051%
7	5.2206%

Table 9.1: P-Coëff efficiency

9.1 Work estimate

If we assume that the innermost sum of Equation 9.7 takes roughly constant time, then we can estimate the total runtime. The total runtime will then mostly depend on how many $(\alpha, \beta : \exists \delta \simeq \beta : \alpha \leq \delta)$ pairs exist. We estimated this by taking one α sample for every size class, and counting the number of betas for further size classes. Figure 9.1 shows the number and distribution if βs for a given α of a specified size. Multiplying the estimated total βs by the number of αs per size class, and summing those gives us a total work estimate of $4.59 * 10^{16}$ (α, β) pairs.

9.2 Computing the P-coefficients

As noted in [6] efficient implementations for $P_{n,k,\alpha,\beta}$ are only known for $k \leq 2$. For $k = 2$ the formula is $P_{n,2,\alpha,\beta} = 2^{C_{n,\beta-\alpha}}$, with $C_{n,\gamma}$ the number of connected components in the boolean function γ. Computing powers of two is easy, it's just a bitshift. But the difficult part is the CountConnected operation.

9.3 CountConnected

In graph theory, for a general graph counting the number of connected components in a graph is a solved problem. Just pick a node, start following edges and adding nodes to the connected component. Once you can't reach another node, then this component is done, and we start again, picking a new unvisited node and repeating until there are no nodes left. For our purposes, this is far too slow. CountConnected

[4] See Section 9.3.7

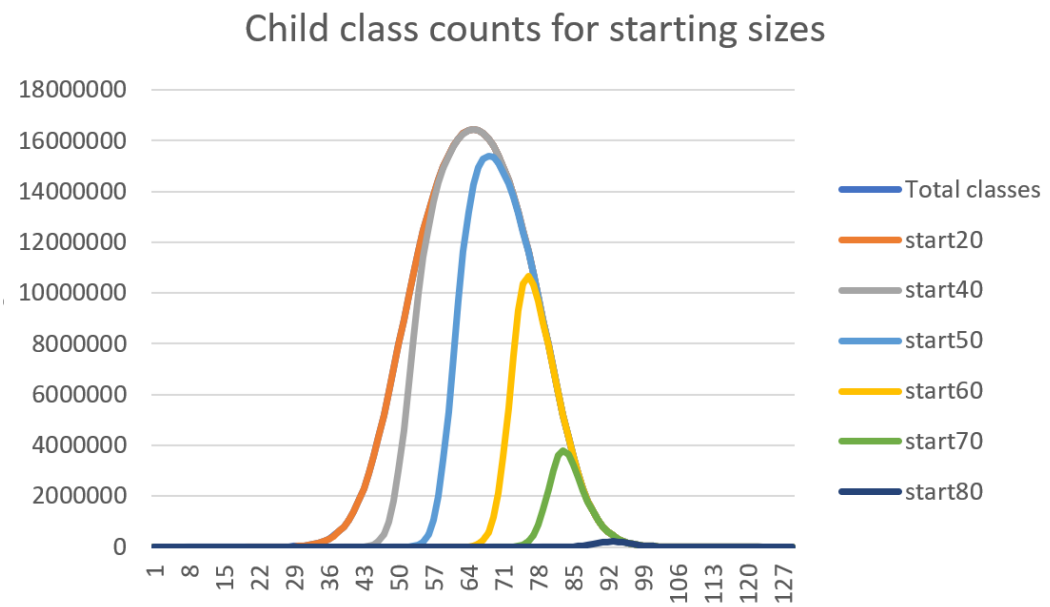

Figure 9.1: Number of Equivalence Classes per size class that have a permutation that is larger than the random sample at the indicated class sizes. Note that the number of child equivalence classes rises incredibly quickly. It takes only 15 steps for most of the equivalence classes to have a permutation larger than the starting MBF. Note that here we list the number of βs *larger* than a given α, that's because I switched the iteration order later on due to implementation concerns. It doesn't really matter, as for the smaller case, we just have to flip the figure along the x-axis. The total work estimate is the same.

has to be run ten quintillion times, so we definitely need to have a fast implementation. Luckily we've got a very specific graph structure and a fixed size, so it should be possible to do better. In fact, we've developed several algorithms that do exactly this, exploiting the structure inherent in boolean functions. Each with their own strengths and weaknesses. While these algorithms are sufficient as we will see in Chapter 10, the hope still remains that it's possible to do better.

9.3.1 Group Merging

The first algorithm is a slight modification of the implementation used in the python implementation of Patrick De Causmaecker's D(8) computation. It works by creating initial groups, of which it is known that all elements below it will be part of the top element's group. Then, the algorithm continually tries to merge groups that are connected until it can't merge any more. At that point we've found our groups.

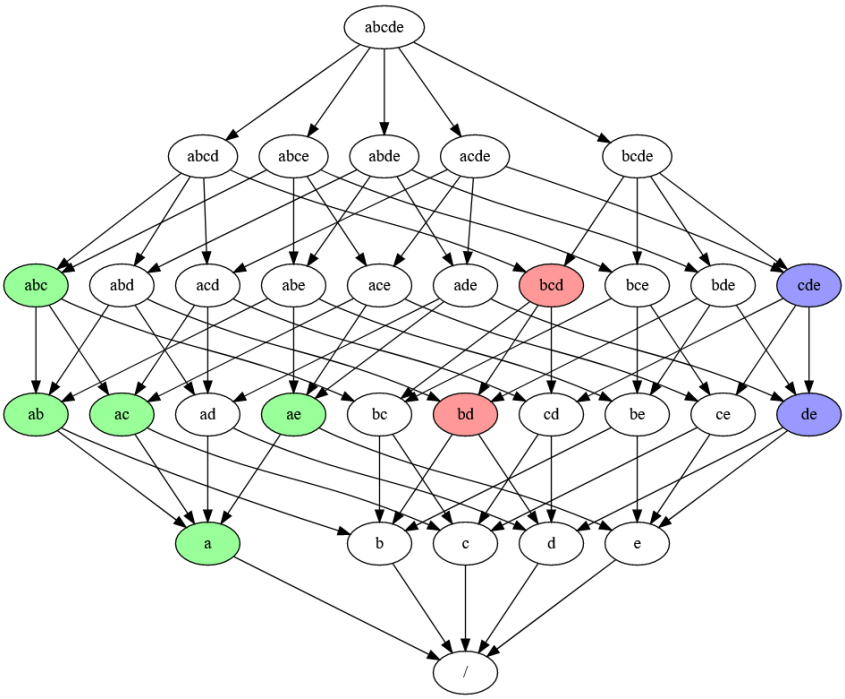

Figure 9.2: Connected components of an example graph. In this case there are 3 connected components.

```
int countConnected(AC top, MBF bot) {
  vector<MBF> groups;
  // create initial groups from the AntiChain top
  for(groupTop : top) {
    if(!bot.contains(groupTop)) {
      groups.push_back(groupTop.monotonizeDown());
    }
  }

  // merge connected groups
  for(int i = 0; i < groups.size(); i++) {
    recheckGroup:
    for(int j = i + 1; j < groups.size(); j++) {
      if(!((groups[i] & groups[j]) <= d)) {
        groups[i] = groups[i] | groups[j]; // merge connected groups
        groups.remove(j);

        // Because we modified group i, all connections with
        // groups i+1 through j have to be rechecked
        goto recheckGroup;
      }
    }
  }
  return groups.size();
}
```

9.3.2 FloodFill

This method is actually quite similar to the 'naive' method, where instead of processing new nodes one by one we perform an agressive 'flood fill'.

Start by picking a seed node for our first group (preferentially in as low a layer as possible). We take all nodes we've gathered in the group so far, and perform a *monotonizeUp* operation. Then we mask with the original graph again so we don't claim nodes we shouldn't. Then we perform a *monotonizeDown* operation, and mask with the original graph again to get nodes we've covered. After every *monotonizeX* and mask operation, we check if the bitset has changed, if it has, then continue. If it hasn't, then we must have covered the entire group. Remove the group from the graph, pick a new seed node and repeat.

```
int countConnected(BF graph) {
  int totalConnectedComponents = 0;

  while(!graph.isEmpty()) {
    // get seed element from an arbitrary group
    // guaranteed to have no down-connections by picking first
    BF expandedDown = graph.getFirst();
    totalConnectedComponents++;
    while(true) {
      // add all elements connected above the current group
      BF expandedUp = expandedDown.monotonizeUp() & graph;
      // if no change, then we've explored the entire group
      if(expandedUp == expandedDown) break;
      // add all elements connected below the current group
      expandedDown = expandedUp.monotonizeDown() & graph;
      // if no change, then we've explored the entire group
      if(expandedUp == expandedDown) break;
    };
    // remove the group we found and start looking for the next one
    graph = graph & ¬expandedDown;
  }
  return totalConnectedComponents;
}
```

9.3.3 Singleton Elimination

This implementation of countConnected works well, but it spends a lot of cycles processing trivial groups that could have been found and eliminated earlier. It occurs quite often that single elements are disconnected from all others, and thus form singleton components. We can actually detect and count these all at once. To detect these singletons, we must look back and forth across every variable. If two nodes are connected across this variable, then they will register each-other as non-singletons.

```
// graph is in/out parameter. After executing the
// given graph will have it's singleton elements removed
// Returns the number of singletons
int countAndRemoveSingletons(BF& graph) {
  BF nonSingletons = BF::empty;
  for(unsigned int v = 0; v < Variables; v++) {
    BF mask = varMask[v];
    int shift = 2^v;
    nonSingletons |= (graph & mask) >> shift; // check above
    nonSingletons |= (graph & ¬mask) << shift; // check below
  }
  int singletonCount = |graph & ¬nonSingletons|;
  graph &= nonSingletons; // remove singletons from graph
  return singletonCount;
}
```

9.3.4 Leaf Elimination

We can greatly improve the performance of the singleton optimisation by also eliminating all leaf elements, that is, all elements only connected to one other node. We must be careful not to remove groups of only 2 elements in their entirity, so we will split this process into first removing all elements with exactly one node below them and none above, and then the reverse, removing all with exactly one above and none below. Doing this first greatly improves the Singleton Elimination from the previous section, and lowers the number of iterations required for the left-over more complex groups.

```
// eliminates all elements that have exactly
// one node above them, and no elements below them.
BF eliminateDownLeaves(BF graph) {
  BF shiftedDown0 = (graph & varMask[0]) >> 1;
  BF occuredAtLeastOnce = shiftedDown0;
  BF blockedElements = BF::empty;

  for(unsigned int v = 1; v < Variables; v++) {
    BF shiftedFromAbove = (graph & varMask[v]) >> 2^v;
    // add elements that have 2 or more connections
    blockedElements |= occuredAtLeastOnce & shiftedFromAbove;
    occuredAtLeastOnce |= shiftedFromAbove;
  }

  // need to also block from below
  for(unsigned int v = 0; v < Variables; v++) {
    BF shiftedFromAbove = (graph & ¬varMask[v]) << 2^v;
    blockedElements |= shiftedFromAbove;
  }

  BF elementsToCull = occuredAtLeastOnce & ¬blockedElements;

  return graph & ¬elementsToCull;
}
```

9.3.5 Direct Method

The hope exists for there to be a direct method for computing countConnected, some fixed series of operations that when executed once, yields the answer. Thus far I have found no efficient implementation, but a hypothetical algorithm for this could be to define a total order over the elements of the graph, and have any element eliminate smaller elements it's directly or indirectly connected to. That way, for every connected component, all elements but the largest one will be removed, and then a simple `popcnt` gives us the number of connected components. Sadly any hypothetical implementations I've been able to come up with, need to be extremely long to deal with the worst-case scenarios (as shown in Figure 9.3), which would negate the gains over the iterative algorithms. Perhaps in hardware this operation could be implemented better, though more work is needed.

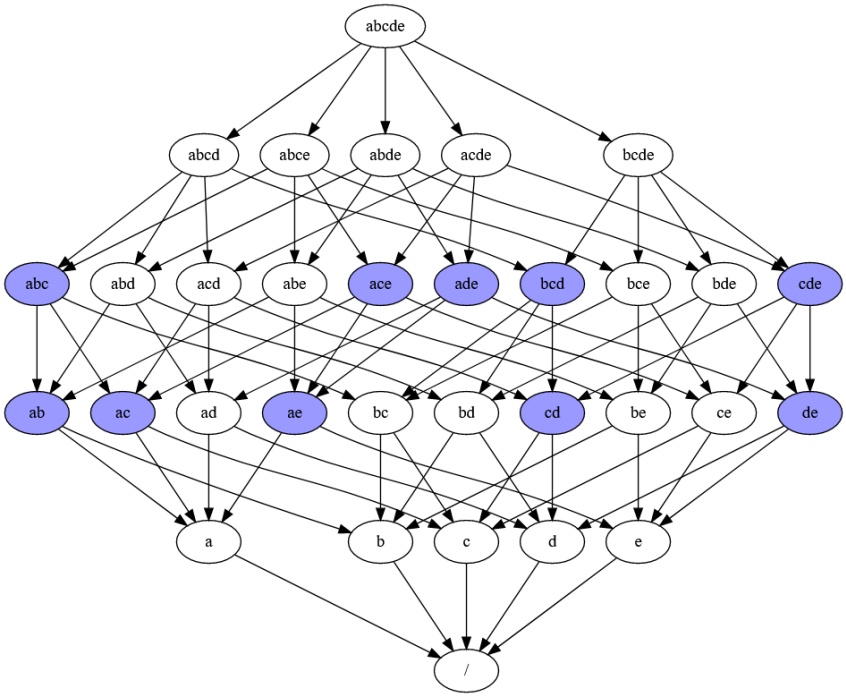

Figure 9.3: A very difficult instance for counting connected components. This whole graph is in fact one long chain (and therefore one connected component), starting at *ab*, to $abc, ac, ace, ae...$ until *bcd*. All known iterative algorithms will have to spend 5-10 cycles exploring this connected component.

9.3.6 Parallelization Troubles

All of these methods have advantages and disadvantages, the main issue is the limits to optimizability. A common optimisation technique is SIMD computation, executing the same instructions for 4, or 8 countConnecteds at a time. The problem with the iterative methods is that they have different loop durations. One countConnected

might take from 3 to 8 cycles as shown in Figure 9.6. This large variation means that any SIMD-like method would suffer from this variability. We'd have to keep iterating until the slowest countConnected in the batch was finished, wasting a lot of valuable computing power. For SIMD the batch size would be 4-8, so this effect would not be too severe, but especially GPU computing would suffer from this, since there computations are usually performed with batches of 32, or even 64 SIMD threads! Perhaps this could be solved with an advanced SIMD implementation, where new countConnecteds are swapped in and out of the working set as they finish. While this is certainly possible using condition masks, the fact that there's so much branching means most of our improvement will be undone, since all of the different code paths have to be executed for every iteration.

9.3.7 Benchmarks

6 Variables

On CPU, all these methods take a similar time for computing D(8). Group Merging takes about 8 minutes. FloodFill without optimizations comes in second at 7 minutes. And with Singleton and Leaf Elimination we can then get down to 6 minutes. All benchmarks were done on my 4-core intel i5 4690k.

7 Variables

For 7 variables doing proper benchmarks is quite a lot harder. We must first build a proper representative benchmark set. For the benchmark set to be representative we have to produce data that mirrors the P-Coëfficient formula's structure. We explored 1/10000 α MBFs, and 1/10000 β MBFs per α. That resulted in 400'000'000 representative (α, β) pairs. Computing this subset alone took 22 hours and 56GB of RAM on a single 36-core Genius Node[23]. Luckily actually running benchmarks using it is quite efficient, and can be done on a smaller home computer.

Method	**6 Variables**	**7 Variables**
Group Merging	124.9ns	249.7ns
FloodFill	66.5ns	107.3ns
FF+Singleton Elimination	53.6ns	87.7ns
FF+SE+LE Down	57.5ns	102.9ns
FF+SE+LE Up	48.3ns	70.4ns

Table 9.2: Benchmark times for CountConnected over 7 variables. All measured single-threaded on an Intel i5-4690k

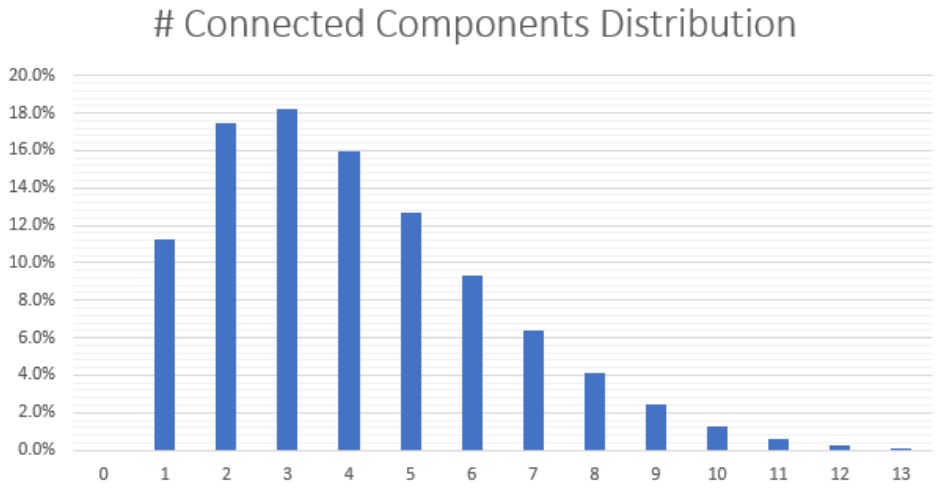

Figure 9.4: Connection Count Distribution. This histogram shows the relative frequenties for the results of countConnected for 7 Variables. Only shown here is a small range, but countConnected can return up to and including 35.

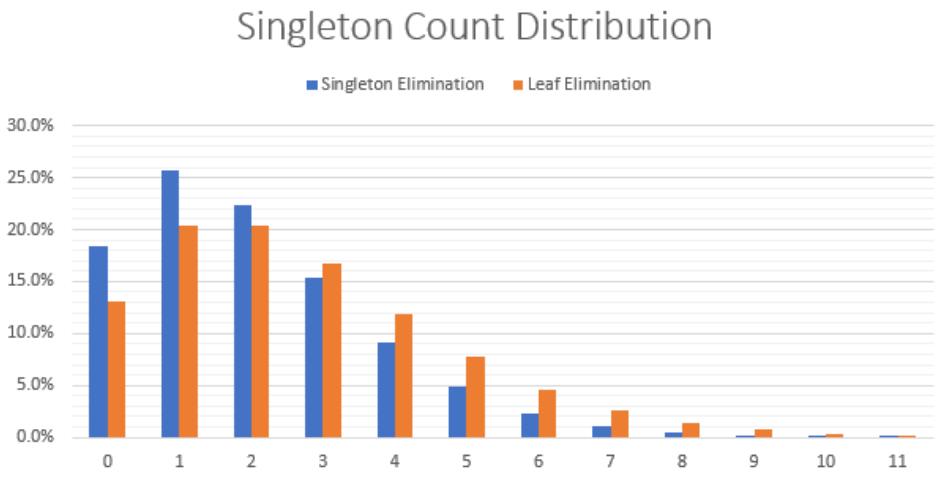

Figure 9.5: Singleton Count Distribution. Compares the improvement in singleton counts of regular Singleton Elimination, with Singleton Elimination following Leaf Elimination.

Method	Singletons	FloodFill Iterations
Basic FloodFill	0	6.967
FF+Singleton Elimination	2.058	4.909
FF+SE+LE Down	2.058	4.909
FF+SE+LE Up	2.670	4.061

Table 9.3: Singleton Counts and remaining FloodFill Iterations for various optimisation strategies. Computed for 7 Variables. On average the number of connected components is 4.07396, so being able to remove 2.67 of those as singletons is a solid improvement.

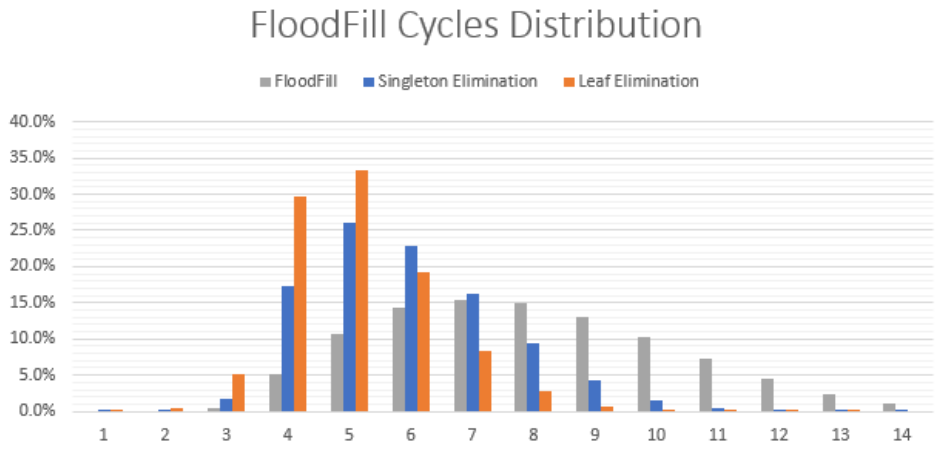

Figure 9.6: Runtime Distribution in Cycles. Shows how many cycles FloodFill still has to run to find the remaining connected components. It is clear that the Singleton Elimination, and especially SE combined with Leaf Elimination reduce the number of cycles substantially. This contraction in the distribution also has a positive effect on the problems regarding SIMD and GPGPU computing.

Chapter 10

FPGA Acceleration

The CountConnected problem, and working with MBFs in general, is very well-suited to FPGA computing. The reason is that pretty much all of our operations involve parallel boolean operations, such as AND and OR, which can be implemented in the logic blocks, or the permuting of bits, which maps perfectly to the logic block interconnects. If we were to move the computation for that innermost sum of Equation 9.7 to a dedicated piece of hardware, then we could greatly improve our performance. We'd split this equation into the computing Equation 10.3 on FPGA, and Equations 10.1 and 10.2 on CPU. It would work by the CPU producing large buffers of α, β pairs, which are then sent to the FPGA over a PCIE link. Once all Psums have been computed, the FPGA sends a buffer back to the CPU, which then adds them together.

$$D(n+2) = \sum_{\alpha \in R_n} |[\bot, \alpha]| D_\alpha \beta_{sum,\alpha} \tag{10.1}$$

$$\beta_{sum,\alpha} = \sum_{\substack{\beta \in R_n \\ \exists \delta \simeq \beta : \alpha \leq \delta}} |[\beta, \top]| \frac{D_\beta}{n!} P_{sum,\alpha,\beta} \tag{10.2}$$

$$P_{sum,\alpha,\beta} = \sum_{\substack{\gamma \in Permut_\beta \\ \alpha \leq \gamma}} 2^{C_{\alpha,\gamma}} \tag{10.3}$$

A basic implementation of the FloodFill CountConnected Operation could be implemented in hardware as shown in Figure 10.1. From that with some input/output queues we can create a module for computing P-Coëfficients shown in Figure 10.2, and creating one of those for every permutation gives us the full circuit shown in Figure 10.3. These figures are mostly illustrative as many optimizations should still be applied (such as Singleton and Leaf Elimination from Sections 9.3.3 and 9.3.4), and not all connections are shown. One such CountConnected core is quite cheap to synthesize. The Largest components will be the *monotonizeUp* and *monotonizeDown* blocks, each of which in my estimation would take 104 Logic Elements each, using the 8-input-4-output ALM modules from the Intel Stratix FPGA family[25]. All

other components are quite a lot smaller, as those don't have to process 128 separate data wires, so we can assume ∼100 extra LE overhead, giving us about 300 Logic Elements per CountConnected Core.
Modern FPGA hardware, such as the Intel Stratix GX 2800[26], run at 300-750MHz[27], and provide 933'000 programmable ALM blocks. Looking back at the FloodFill cycles (Table 9.3) shows us that with Singleton and Leaf elimination, we could get down to 4 cycles per P-Coëfficient on average. At 500MHz, that's 8ns. Compare that to the 70ns it takes one CPU core to compute one such P-Coëfficient (Table 9.2) and the advantage of FPGA computing becomes clear. Add to that the fact that one CPU may only have 14-32 cores, whereas we could synthesize this P-Coëfficient block 1000-2000 times on a single Stratix 2800 FPGA. This is the crux of the FPGA argument, *one* FPGA could perform the computational equivalent of ∼500 Intel Xeon CPUs, at about half a billion (α, β) pairs per second. (which incidentally, is quite close to the amount of bandwidth a PCIE3 16x bus can provide.) With a total work of $4.59 * 10^{16}$ (α, β) pairs[1], we'd need 10^8 FPGA seconds, or 38 FPGA months.
Sadly, this is where the bad news comes. First of all, we have to perform the whole computation twice to make sure no electrical noise or other disturbances in the FPGAs affected the results. We can have a simple check for this by storing and then comparing each of the $\beta_{sum,\alpha}$ values to check whether there were any errors during computation, and if needed, rerun the erroneous computations on CPU. Secondly, there is barely any existing FPGA supercomputing hardware. There's a prominent supercomputer at Paderborn University called Noctua[28], which has 32 520N[29] FPGA cards. Because the problem scales very well, it should be possible to perform the entire computation twice in about 2.5 months.
2.5 months might be a bit too long to use Paderborns FPGA supercomputer, so another option would be to purchase our own hardware. Luckily it seems that the FPGA High Performance Computing industry is really heating up these last few years, with large improvements every year. Recently, two very large and powerful FPGAs became available, namely the Intel Stratix GX 10M [26] (10.2M LE), and Xilinx' VU19P [30] (9.8M LE), which nearly quadruple the number available logic units. Newer FPGA cards such as Bittware's IA-840F card using the Agilex AGF 027[31] boast PCIE4 compatibility, with double the bandwidth and greater clock speeds, so if it isn't feasible with today's hardware, it will be with tomorrow's.

[1] See Section 9.1

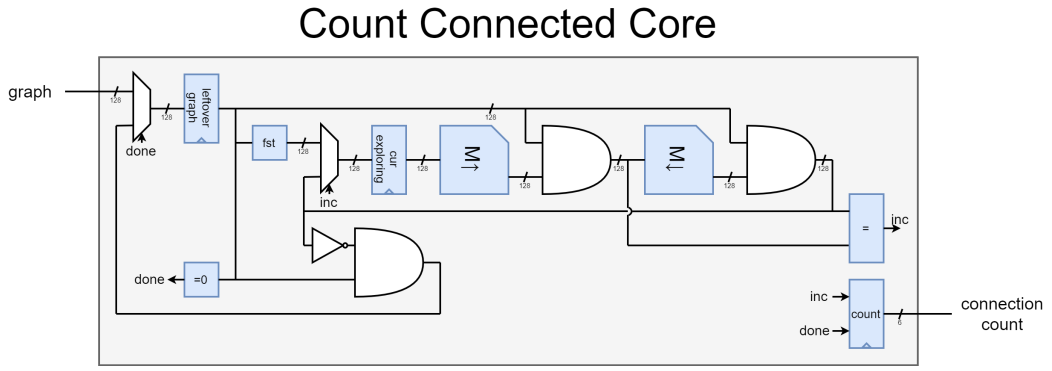

Figure 10.1: CountConnected Core

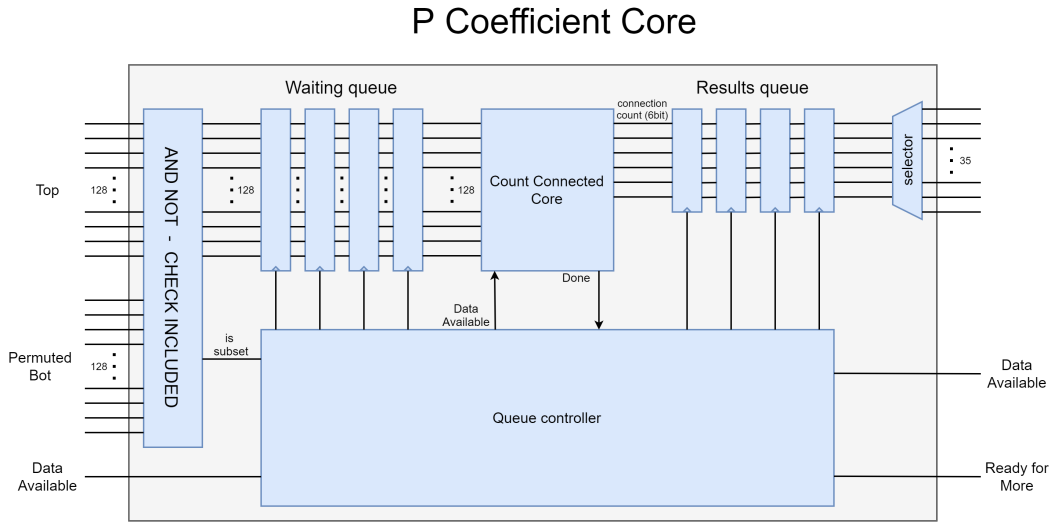

Figure 10.2: PCoëff Core

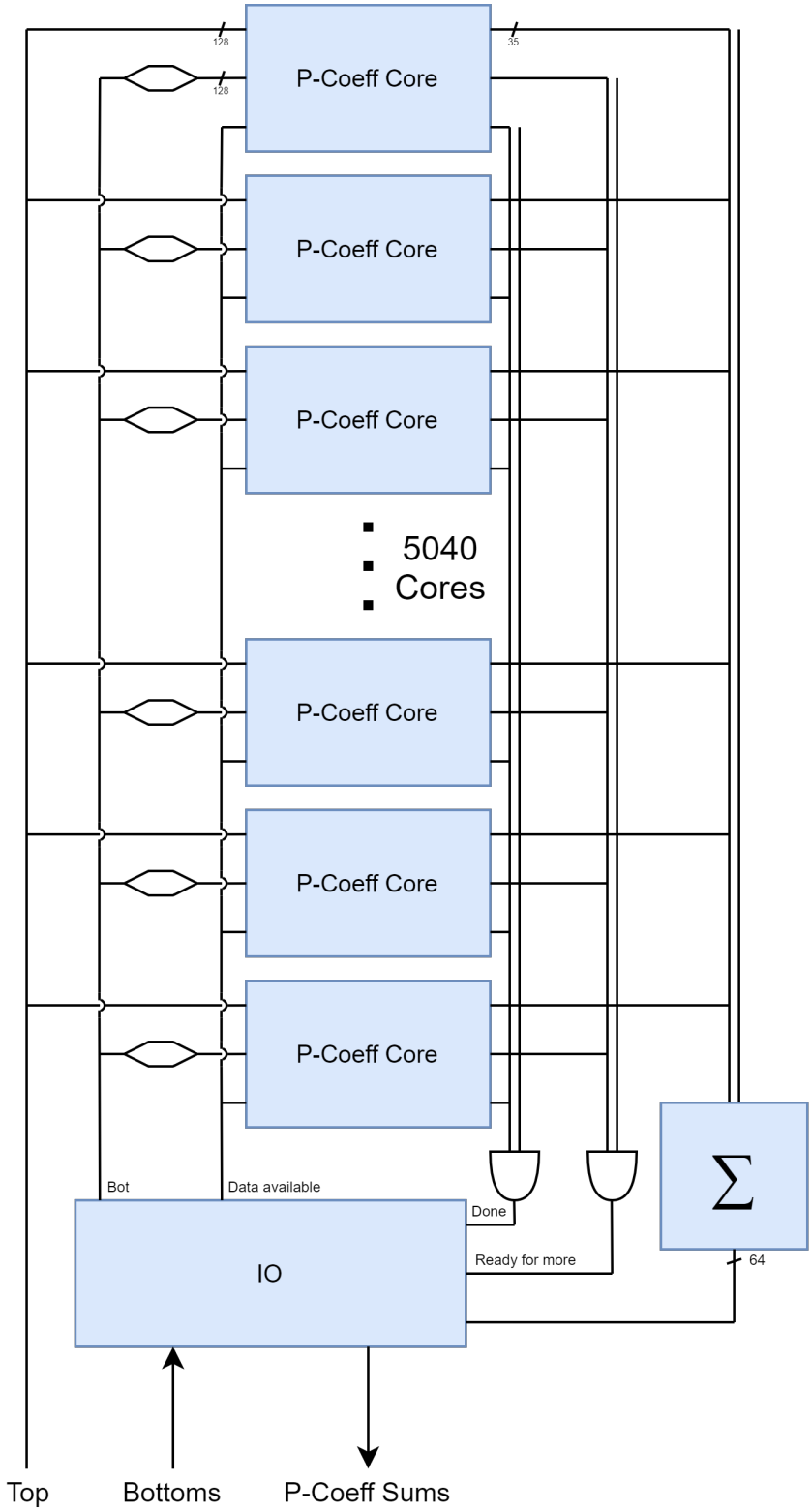

Figure 10.3: Example naive parallel PCoëff processing.

Chapter 11

Discussion and Conclusion

Enumerating all Equivalence Classes R(7) has been done before[32], but they needed 8 CPU Years to compute it, parallelized over several machines it took them about a year. Whereas the new and highly optimized method described in this thesis takes just 10 minutes on a 4-core intel i5. This is a major improvement, one which will hopefully enable wider use in the scientific community. The fact that the highest known R number is now computable in such short time gives hope that, computation of R(8) is not far off. To estimate: The complexity is approximately linear in $R(N)$, which will be at least $D(N)/N! = 1.4 * 10^{18}$. If we assume that the time per element is equal (quite optimistic) then the resulting time is 900 CPU years. On a large supercomputer such as Breniac[33] this might be possible, though it would probably still take around a year. For the time being, it's infeasible. And that doesn't even take into account the massive memory requirement to store all equivalence classes of the current size, that would be around 20PB. The whole dataset would be about 500PB if we wished to store it entirely.

Computing all interval sizes $|[\bot, \alpha]|$ has never been done for 7 variable MBFs, so this is an entirely new achievement. It's even fast enough that it doesn't even require a supercomputer, being computeable on a modern computer in 36 hours. This aspect is especially interesting for computing variations (starting from a different bottom) so using these intervals in other research will also be possible. While there is no publication that has done this, I can compare to one implementation: What prompted my promotor to start this thesis was that he received an e-mail from a student in germany, who claimed to have a method for computing these interval sizes, though his computing cluster was still working on it. At the time of writing, as far as I know he has not finished his calculation yet.

The last Dedekind Number found was D(8) in 1991 by Doug Wiedemann. [2] At the time, it took him 200 hours on the Cray-2 supercomputer. Today, many recomputations of D(8) have been performed, on newer hardware too, but using mostly similar methods. In his bachelor thesis, Arjen Teijo Zijlstra implemented the same method in C++, and computed it in 3.5 hours using 12 cores. [34]. Patrick De Causmaecker did it with his Java implementation in 8 hours on a 4-core machine using his P-Coëfficient method, which is a variation upon the method Wiedemann used. [6].

My implementation of the P-Coëfficient method using the `EquivalenceClassMap` data structure computes D(8) in 6 minutes, on a 4-core intel i5. I've also been able to compute D(8) using a different method as a verification for the computed intervals and symmetry counts, taking only 76 seconds single-threaded [1] (though this requires having computed all 7-variable intervals and symmetries first). For D(9) however, it seems infeasible on CPU supercomputers, as it is estimated to take more than 8 months on Breniac[33], in the ideal case. It might be possible to reduce this time using GPU computing, though the core operation (countConnected) does not translate well to the SIMD structure of a GPU.

FPGA supercomputing however, does seem to be the answer. The countConnected operation is uniquely well suited to a hardware implementation. While one FPGA can outperform hundreds of CPUs for countConnected, the fact that FPGA supercomputing is still not widespread means our available computing hardware is still fairly limited. I estimate it would take about two months to do the full calculation on the FPGA supercomputer Noctua at Paderborn University. [28] This is already much more feasible, because it's a much smaller supercomputer than Breniac, it could be used for longer with a lower budget. And even if it's too long, or if some unforeseen implementation detail appears that slows the computation down, the explosion of FPGA computing capability these last few years means using them to compute D(9) is not far off anyway.

[1] See section 8.5

Chapter 12

Future Work

Perhaps further progress can be made on a direct method for countConnected. If it's possible to do it efficiently, then it might become feasible to compute D(9) on large GPU clusters, avoiding the need for expensive FPGA hardware.

In the coming months, and during my PhD, we intend to do the actual FPGA implementation, perhaps in collaboration with Paderborn University, or using purchased hardware. With these improvements and especially with FPGA hardware, computing D(9) is finally on the horizon. It's no longer a question of if, but when.

There may also still be some hope for computing R(8) on a very large (probably Tier 1 or even 0) supercomputer, by finding an algorithm that can iterate over all equivalence classes without needing the large memory consumption, similar to how large memory usage was sidestepped in Section 5.6, and finding an incremental canonization algorithm to improve the performance of this critical operation.

Chapter 13

Acknowledgements

The resources and services used in this work were provided by the VSC (Flemish Supercomputer Center), funded by the Research Foundation - Flanders (FWO) and the Flemish Government.

Bibliography

[1] R. Dedekind. *Über Zerlegungen von Zahlen Durch Ihre Grössten Gemeinsamen Theiler*, pages 1–40. Vieweg+Teubner Verlag, Wiesbaden, 1897.

[2] Doug Wiedemann. A computation of the eighth dedekind number. https://link.springer.com/article/10.1007%2FBF00385808, 1991.

[3] Matthias Vandersanden Lennart Van Hirtum. Physics3d library. https://github.com/ThePhysicsGuys/Physics3D.

[4] caclcrypto. uint256_t library. https://github.com/calccrypto/uint256_t.

[5] Patrick De Causmaecker and Stefan De Wannemacker. Partitioning in the space of antimonotonic functions, 2011.

[6] Patrick De Causmaecker and Stefan De Wannemacker. On the number of antichains of sets in a finite universe, 2014.

[7] Patrick De Causmaecker, Stefan De Wannemacker, and Jay Yellen. Intervals of antichains and their decompositions, 2016.

[8] Tilman Piesk. Dedekind number — Wikipedia, the free encyclopedia. https://en.wikipedia.org/wiki/Dedekind_number, 2021. [Online; accessed 27-April-2021].

[9] The Computer Language Benchmarks Game. C++ g++ versus java fastest programs. https://benchmarksgame-team.pages.debian.net/benchmarksgame/fastest/gpp-java.html. [Online; accessed 31-May-2021].

[10] Microsoft. Visual studio. https://visualstudio.microsoft.com.

[11] Matt Godbolt. Compiler explorer. https://godbolt.org.

[12] Matt Godbolt. What has my compiler done for me lately? unbolting the compiler's lid. https://www.youtube.com/watch?v=bSkpMdDe4g4, CppCon 2017.

[13] Andrei Alexandrescu. Optimization tips - mo' hustle mo' problems. https://www.youtube.com/watch?v=Qq_WaiwzOtI, CppCon 2017.

[14] Jason Turner. Practical performance practices. https://www.youtube.com/watch?v=uzF4u9KgUWI, CppCon 2016.

[15] Chandler Carruth. Efficiency with algorithms, performance with data structures. https://www.youtube.com/watch?v=fHNmRkzxHWs, CppCon 2014.

[16] John Lakos. Local ('arena') memory allocators. https://www.youtube.com/watch?v=nZNd5FjSquk, CppCon 2017.

[17] Jonathan Müller. Writing cache friendly c++. https://www.youtube.com/watch?v=Nz9SiF0QVKY, Meeting C++ 2018.

[18] Ansel Sermersheim. Multithreading is the answer. what is the question? https://www.youtube.com/watch?v=GNw3RXr-VJk, CppCon 2017.

[19] Herb Sutter. Atomic weapons. https://www.youtube.com/watch?v=A8eCGOqgvH4, C++ and beyond 2012.

[20] Herb Sutter. Lock-free programming (or, juggling razor blades). https://www.youtube.com/watch?v=c1gO9aB9nbs, CppCon 2014.

[21] Timur Doumler. Want fast c++? know your hardware! https://www.youtube.com/watch?v=BP6NxVxDQIs, CppCon 2016.

[22] Intel. Intrinsics guide. https://software.intel.com/sites/landingpage/IntrinsicsGuide/. [Online; accessed 8-May-2021].

[23] Vlaams Supercomputing Centrum. Genius cluster information. https://vlaams-supercomputing-centrum-vscdocumentation.readthedocs-hosted.com/en/latest/leuven/tier2_hardware/genius_hardware.html, 2014. [Online; accessed 13-May-2021].

[24] Bianca Schroeder, Eduardo Pinheiro, and Wolf-Dietrich Weber. Dram errors in the wild: A large-scale field study. *Commun. ACM*, 54(2):100–107, February 2011.

[25] Intel. Stratix alm module diagram. https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/stratix-10/ug-s10-lab.pdf, 2019. [Online; accessed 4-May-2021; Page 11].

[26] Intel. Stratix gx 2800 product sheet. https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/pt/stratix-10-product-table.pdf, 2019. [Online; accessed 4-May-2021].

[27] Tian Tan, Eriko Nurvitadhi, David Shih, and Derek Chiou. Evaluating the highly-pipelined intel stratix 10 fpga architecture using open-source benchmarks. In *2018 International Conference on Field-Programmable Technology (FPT)*, pages 206–213, 2018.

[28] Paderborn University. Noctua cluster information. https://pc2.uni-paderborn.de/hpc-services/available-systems/noctua/, 2018. [Online; accessed 4-May-2021].

[29] Intel. 520n pcie fpga board. https://www.bittware.com/fpga/520n/, 2019. [Online; accessed 4-May-2021].

[30] Xilinx. Vu19p product sheet. https://www.xilinx.com/support/documentation/data_sheets/ds923-virtex-ultrascale-plus.pdf, 2020. [Online; accessed 4-May-2021].

[31] Intel. Agilex series product sheet. https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/pt/intel-agilex-f-series-product-table.pdf, 2020. [Online; accessed 3-June-2021].

[32] Tamon Stephen and Timothy Yusun. Counting inequivalent monotone boolean functions, 2012.

[33] Vlaams Supercomputing Centrum. Breniac cluster information. https://vlaams-supercomputing-centrum-vscdocumentation.readthedocs-hosted.com/en/latest/leuven/tier1_hardware/breniac_hardware.html, 2014. [Online; accessed 4-May-2021].

[34] Arjen Teijo Zijlstra. Calculating the 8th dedekind number. https://fse.studenttheses.ub.rug.nl/11075/1/ThesisDedekind-ArjenZijlstra.pdf, 2013. University of Groningen, Bachelor Thesis.