EXPERIMENTAL MATHEMATICS TECHNIQUES FOR BOOLEAN FUNCTIONS AND COMBINATORIAL GAMES

By

BLAIR SEIDLER

A dissertation submitted to the

School of Graduate Studies

Rutgers, The State University of New Jersey

In partial fulfillment of the requirements

For the degree of

Doctor of Philosophy

Graduate Program in Mathematics

Written under the direction of

Doron Zeilberger

And approved by

New Brunswick, New Jersey

October, 2023

ABSTRACT OF THE DISSERTATION

Experimental Mathematics Techniques for Boolean Functions and Combinatorial Games

By BLAIR SEIDLER

Dissertation Director:

Doron Zeilberger

The theme of this dissertation is the application of experimental techniques to the analysis of Boolean functions and combinatorial games. The three projects comprising this paper all employ Zeilberger's overlapping stages methodology to build successively more efficient algorithms to mitigate the exponential growth of the combinatorial objects being studied.

The first two projects concern Boolean functions. We produce a catalog of minimal functions for Boolean circuits of a small number of variables using straight-line programs as our representation of circuits. We then analyze the statistical moments of subcubes contained in Boolean functions, extending the work of Thanatipanonda. We also analyze mixed moments and the correlation between numbers of subcubes of various sizes.

The third project analyzes the combinatorial game Juniper Green along with several new variations. Lemoine completely solved the problem of which initial positions of Juniper Green are winnable by the first player. We consider the Sprague-Grundy values of initial positions in the original game and variations.

ACKNOWLEDGEMENTS

I am extremely grateful to my advisor, Doron Zeilberger, for his support and guidance over the past several years. He introduced me to experimental mathematics, suggested many interesting problems to work on (three of which became the content of this dissertation), gave me invaluable advice and support during our research meetings, and always knew when to encourage me and when to challenge me. Perhaps as importantly, he shared his genuine enjoyment of mathematical exploration with me.

I also want to thank the other members of my committee for their contributions to my development as a mathematician. Bhargav Narayanan taught me much of what I know about combinatorics during several courses, many talks, and informal discussions around the department. Michael Saks graciously spent a year discussing computational complexity with me every week despite the time demands of serving as department chair. Thotsaporn Aek Thanatipanonda wrote the 2020 paper which inspired Chapter 3 of this document. I thank them all for their support and for serving as my committee.

I want to thank the entire Mathematics Department at Rutgers for providing a supportive and friendly environment in which to study, teach, and learn. It was this sense of community which convinced me to enroll at Rutgers, and the last five years have confirmed that the positive atmosphere I felt when I visited is real. There are too many people who have supported me to list all of them, but some of the current and former members of the department I particularly wish to thank are: AJ Bu, Matt Charnley, Yael Davidov, Robert Dougherty-Bliss, Quentin Dubroff, Mariano Echeverria, Rashmika Goswami, Katie Guarino, Zheng-Chao Han, Edna Jones, Jeff Kahn, Mariusz Mirek, David Molnar, Shubhangi Saraf, Chloe Wawrzyniak, Chuck Weibel, and Corrine Yap.

I want to thank the Rutgers Academy for the Scholarship of Teaching and Learning for being an inspiring and supportive group of educators and scholars. This group helped me connect to the larger Rutgers community and gave me opportunities to participate in many valuable workshops and discussions about higher education.

Finally, I want to thank my whole family, particularly my spouse, Jenn, and my children, Kat and Elizabeth, for their love and support during this journey. From the moment I started thinking about leaving my job as a high school teacher through the completion of this dissertation, you have encouraged me at every turn. I could not have done this without you.

Contents

Foreword **x**

1 Introduction **1**

2 Minimal Circuits for Boolean Functions **5**

2.1 Introduction 5

2.1.1 Definition and Representation of Boolean Functions 6

2.1.2 Equivalence Classes 9

2.1.3 Straight-Line Programs 10

2.2 General Functions of Four or Fewer Variables 12

2.2.1 Methodology 12

2.2.2 Results 16

2.2.3 Future Project 18

2.3 Monotone Functions of Five or Fewer Variables 20

2.3.1 Definitions and Notation 20

2.3.2 Methodology 21

2.3.3 Results 22

3 Moments of the Number of Subcubes in Boolean Functions **24**

3.1 Introduction 24

3.1.1 Definitions and Notation 25

3.2 Previous Results 26
3.3 Methodology 27
3.3.1 Data Structure 28
3.3.2 Kernels and Remainders 29
3.4 Moments of Numbers of 1-dimensional Subcubes 30
3.5 Moments of Numbers of 2-dimensional Subcubes 32
3.6 Moments of Numbers of 3-dimensional Subcubes 33
3.7 Mixed Moments 34
3.7.1 Correlation 36

4 Combinatorial Game - Juniper Green 39

4.1 Introduction 39
4.1.1 Combinatorial Games - Definitions 40
4.1.2 Position Graphs 41
4.1.3 Sprague-Grundy Values 43
4.2 Juniper Green 44
4.2.1 Winning Strategies 45
4.3 Methodology 46
4.3.1 Traditional Game - “Hard Version” 47
4.3.2 Additive Version 50
4.4 Original Game 53
4.4.1 Pruning Attempt 53
4.4.2 Results 56
4.5 Factor Restriction Variation 56
4.5.1 Symmetric Cases 57
4.5.2 Asymmetric cases 59
4.6 Additive Variations 59
4.6.1 Equal Singleton Sets 60

4.6.2 Unequal Singleton Sets 63

4.6.3 Sets of the Form {1,k} 64

4.6.4 Sets of the Form {2,k} 65

4.7 Future Work 65

5 Conclusions 67

A Catalog of Circuits 69

A.1 Functions of 1 variable 69

A.2 Functions of 2 variables 70

A.3 Functions of 3 variables 70

B Maple Code for Boolean Functions 74

B.1 General Boolean Functions 74

B.2 Monotone Functions 80

C Maple Code for Subcubes of Boolean Functions 83

D Maple Code for Juniper Green 84

List of Figures

4.1 The subgraph containing primes p, q, r 46
4.2 Adjacency graph for $n = 8$ 54
4.3 Reduced adjacency graph for $n = 8$ 55

List of Tables

2.1 Function numbers for Boolean functions of 2 variables. 8
2.2 Straight-line program line formats. 11
2.3 Number of BEC's by circuit complexity 16
2.4 Number of functions by circuit complexity 17
2.5 Number of BEC's by monotone circuit complexity 22

4.1 Output of `Mhgraph(10)` 49
4.2 Output of `Magraph(10,{1,2},{1,2})` 52
4.3 Sprague-Grundy Values with $A = \{1\}, B = \{b\}, b > 1$ 63
4.4 Sprague-Grundy Values with $A = \{2\}, B = \{b\}, b > 2$ 64
4.5 Sprague-Grundy Values with $A = B = \{1, k\}$ 64
4.6 Sprague-Grundy Values with $A = B = \{2, k\}$ 65

Foreword

Many of my close family members have advanced degrees in the physical or biological sciences. When I told them that I was going to focus on Experimental Mathematics, they all gave me quizzical looks. “How can you experiment on something that doesn’t exist?” they asked. I understood their skepticism. As my illustrious advisor has been known to say, “Experimental Mathematics used to be considered an oxymoron.” But the abstract nature of mathematical objects does not exempt them from the scientific method!

What are the fundamental principles of experimentation which underlie the scientific method? The scientist observes some interesting phenomenon, wonders what causes it, and starts to reason why it should be so. After establishing some plausible explanation, they design an experiment with which they will obtain evidence either supporting or refuting that explanation. The hypothesis is updated with the new information, and subsequent experiments are designed and implemented. If all goes well, a satisfactory explanation for the observed phenomenon now becomes a first-author paper, a Theory, or possibly even a Law.

Why should mathematics be so different? We mathematicians also observe phenomena, some relating to tangible real-world objects and some admittedly much more abstract. We wonder why it should be so. “There must be a pattern!” we rant to nobody in particular. But this is where our experimentation comes into play. We can program our computers to generate many cases. Sometimes, this first round of results is enough for us to see what is going on. Sometimes, we need to rely on the computer to help us analyze its own output. But eventually, if all goes well, we have a pattern, an explanation, a conjecture, a theorem...

I suppose that herein lies the difference. In mathematics, absolute certainty is not only possible, but desirable and common. Experimental methods in mathematics frequently provide the ideas that drive a more traditional paper-and-pencil (more likely keyboard-and-LaTeX) proof. Other times, we can build on early rounds of experimentation to generate a computer-aided symbolic proof. Sometimes (as in the original proof of the Four-Color Theorem), the computer's analysis of all of the cases IS the proof.

But sometimes the best we can do is to show that our conjecture is true for however many cases our computer can process in a reasonable amount of time. We may also have evidence beyond those cases that what we think is happening continues to be true. There may even be disagreement among mathematicians about whether this evidence constitutes a sufficiently rigorous proof to promote a conjecture to a theorem.

This last case may well be the place where mathematics and the biological and physical sciences seem most alike. Mathematicians generally have great respect for a long-standing conjecture, supported by a host of supporting evidence, which nobody has been able to disprove or significantly improve upon. To me, that sounds like what scientists mean when they use the word Theory.

Chapter 1

Introduction

"Then, about 2300 years ago came a fellow called Euclid, and Euclid ruined mathematics by turning it into a deductive science." - Doron Zeilberger

The origins of mathematics among prehistoric humans may well have started with simple distinctions such as singular versus plural, large versus small, and curved versus straight. Our distant ancestors then discovered bijections (there are as many rocks in this pile as I have fingers on both hands) and ordinals (I can place the largest rock here, then the next largest, all the way down to this pebble). As humans looked for explanations for how the world worked, they developed primitive religions. The rituals of those religions may well have led to the advent of counting as an aid to ordering the components of the ritual. [[BM11]] Desire to predict upcoming events required inductive reasoning – record what has happened, try to find a meaningful pattern, guess what is going to happen next, wait to see whether you were correct, repeat and refine. Seasons, weather, and astronomical events all seemed to follow some sort of pattern. The wise men and shamans were able to find the order in the chaos, at least some of the time.

As the millennia rolled on, humans developed arithmetic and algebra to handle commerce and inheritance, geometry to aid with agriculture and architecture, and other branches of mathematics as they were needed to solve practical problems. Later, philosophers attempted to impose their particular brand of order on mathematical thought, insisting that there should be some set of absolute truths from which all of the other concepts could be deduced. Modern mathematicians, for the most part, consider the Zermelo-Fraenkel set theory axioms to be that set of absolute truths from which all other mathematical concepts flow, and to which they are somehow subordinate. But even as we produce rigorous deductive proofs of our theorems, there is a great deal of pattern-seeking at the core of what we do.

> Mathematics is not a deductive science – that's a cliché. When you try to prove a theorem, you don't just list the hypotheses and then start to reason. What you do is trial and error, experimentation, guesswork. You want to find out what the facts are, and what you do is in that respect similar to what a laboratory technician does, but it is different in its degree of precision and information. Possibly philosophers would look on us mathematicians the same way as we look on the technicians, if they dared. [[Hal85], p. 321]

This dissertation is first and foremost an exploration of several techniques of experimental mathematics. The several projects contained herein have that essence of inductive reasoning which is at the heart of experimentation. The experimental mathematician employs computer programs to perform more calculations than could reasonably be accomplished by hand. Even so, the objects being studied exhibit exponential growth in some fashion, rendering brute force enumeration powerless at a relatively early stage. Only by observing patterns and finding ways to prune the number of objects under consideration are we able to progress beyond those cases.

One technique common to all three of the following chapters is the methodology of overlapping stages suggested by Zeilberger. [[Zei04]]. In this paradigm, we first employ computer programs which directly encode the definitions of the objects we are studying. These programs are generally quite slow and are only able to compute the smallest cases of the problem, but they have the advantage of being relatively easy to check for accuracy. We then try to find some clever idea which allows us to write a faster program using algorithms which are somewhat removed from the original definition of the problem. We hope that the clever algorithm will run enough faster than the original version that we can extend the number of cases that we can compute in whatever amount of time we consider reasonable. We then confirm that the answers for the smaller cases match those generated by the simpler program, providing some evidence of the validity of the new program. If all of this proceeds well, we may iterate this process several times and be able to generate enough data to yield the solution we are seeking.

In Chapter 2, we explore circuits for computing Boolean functions of small numbers of variables. Our primary goal is to produce a catalog of minimal circuits for all Boolean functions of up to four variables. Our secondary goal is to produce a catalog of minimal circuits using only AND and OR gates for (positive) monotone functions of up to five variables. For any given Boolean function, finding *a* circuit to compute it is easy: we can translate the set of true points into a full disjunctive normal form expression, then build a circuit which computes that expression. We can also find a reasonably efficient circuit by reducing the DNF using the Quine-McCluskey method before constructing the circuit. Finding a minimal circuit presents more of a challenge, because we also need to show that no smaller circuit will work. The number of possible circuits grows exponentially in the number of gates, so a full enumeration becomes impractical quickly.

In Chapter 3, we shift our perspective to focus on Boolean functions as subsets of the discrete n-dimensional unit cube. We explore the statistical moments of the number of subcubes of each dimension contained within those subsets, extending the work of Thanatipanonda [Tha20]. We derive a formula for the mixed moment $\mathbb{E}[X_r X_s]$, and explore several of these mixed moments. We also analyze the asymptotic behavior of the correlation between numbers of subcubes of distinct sizes. In all of these situations, the number of functions we need to consider is 2^{2^n}, where n is the number of variables. We use a combination of the symmetries of the orientations of subcubes and symbolic computation of the number of locations within the discrete n-cube to slow the exponential growth to a manageable level.

In Chapter 4, we move to the field of combinatorial game theory, focusing our attention on the game Juniper Green. Julien Lemoine elegantly solved the winnability of the original game for all values of n, the number of spaces on the game board. [[Lem22]] We consider the Sprague-Grundy values of the game for different values of n. We also introduce several variations of the game, producing Sprague-Grundy values for the initial positions. Several of these variations produce periodic sequences, which we can verify from first principles for some cases.

Chapter 2

Minimal Circuits for Boolean Functions

2.1 Introduction

"Boolean functions, meaning $\{0,1\}$*-valued functions of a finite number of* $\{0,1\}$*-valued variables, are among the most fundamental objects investigated in pure and applied mathematics."* Crama and Hammer [[CH11], p. *xv*]

Mathematicians (including theoretical computer scientists) have spent a great deal of time and effort considering the asymptotic circuit complexity of Boolean functions as the number of variables increases. One desirable result of such explorations would be to find a class of functions representing a language in **NP** for which the asymptotic circuit complexity is superpolynomial. This discovery would prove that $\mathbf{P} \neq \mathbf{NP}$ and resolve perhaps the most perplexing open problem in theoretical computer science.

While this is a worthy endeavor, the result has proven to be elusive. Paul [[Pau77]] and Blum [[Blu84]] used an excellent gate-elimination argument to achieve lower bounds for carefully constructed functions. They used similar inductive arguments in which each stage of the induction fixed the value of a single variable in order to eliminate some number of gates. But these heroic constructions were only able to produce a linear lower bound ($2.5n - O(1)$ for Paul and $3n - O(1)$ for Blum). It has recently been shown that gate-elimination can never produce a superlinear bound [[GHKK18]].

We choose instead to investigate the other extreme of circuit complexity, circuits for Boolean functions of a small number of variables. One of the attractions of the small, finite cases is that we should be able to understand them completely. There are only $65,536$ Boolean functions of four variables, which makes it practical to enumerate them on any modern computer. Our primary objective is to find the circuit complexity of every Boolean function of four or fewer variables. We also investigate the class of monotone functions and produce a catalog containing circuits which will compute each function.

2.1.1 Definition and Representation of Boolean Functions

Definition 2.1. Let $K = \{-1, 1\}$. A *Boolean function* of n variables is a function $f : K^n \to K$.

We deviate from the traditional notation $K = \{0, 1\}$ in part because the labels are irrelevant. The two elements of K are abstract labels, the former representing *false* and the latter representing *true*. In practice, we use these labels to simplify an implementation of the Quine-McCluskey algorithm [[Qui52]] for minimizing Boolean functions, also known as "the method of prime implicants." In this implementation, a zero in a particular position in a prime implicant indicates that the corresponding variable can be either true or false. Using 0 for the wildcard in the Maple implementation is much easier than the traditional * for technical reasons.

We represent Boolean functions in several ways in our code and its output depending on the context. The simplest form is as the set $F = \{v \in K^n : f(v) = 1\}$. This is the set of "true points", easiest to visualize as the subset of the Hamming cube at which the function evaluates to *true*. One advantage of this representation is that it provides both a simple way to evaluate the function at a point v (by checking the condition $v \in F$). It also confirms that the number of Boolean functions of n variables is 2^{2^n}, the size of the power set of K^n.

The second representation is as a set of vectors in $\{-1, 0, 1\}^n$. This representation is used in our implementation of the Quine-McCluskey algorithm mentioned earlier in this section. As an example, the two functions

$$\{[1, 0, -1, 0]\} \quad \text{and} \quad \{[1, -1, -1, -1], [1, -1, -1, 1], [1, 1, -1, -1], [1, 1, -1, 1]\}$$

are equivalent. Several of the functions in our code support this representation of functions. For example, the function `EvalSLPn`, which evaluates a straight-line program on a given input, supports input vectors containing zeroes. It does so by replacing the first zero in the input vector by each of -1 and 1, making a recursive call on each of those vectors, and returning *true* if either of those assignments returns *true*.

The third representation, used only internally in our Maple code, is functionally equivalent to the first. Each vector in the set of true points is converted to an integer between 1 and 2^n by considering the vector as a binary number (using the traditional 0 for *false*) and adding one. The motivation for this representation is efficiency in permuting and negating the variables. Arrays for the permutation and negation of variables can be initialized and stored, and then each member of the set can be shifted quickly using these arrays without having to treat each bit separately. We add one to the binary values for the mundane technical reason that Maple array indices are constrained to start at one. Yes, this constraint makes us long for the more civilized arrays of the C programming language which have indices starting at zero as nature intended.

The final representation of Boolean functions is as a single integer. This is the most compact representation, and it was implemented to be consistent with OEIS sequence A227723 [[OEISd]]. As shown in Table 2.1, the input strings are arranged from least to greatest when interpreted as binary numbers. The number of the function is assigned by reading down the column of truth values, and again interpreting as a binary number.

Table 2.1: Function numbers for Boolean functions of 2 variables.

Input	f_0	f_1	f_2	f_3	f_4	f_5	f_6	f_7	f_8	f_9	f_{10}	f_{11}	f_{12}	f_{13}	f_{14}	f_{15}
$\{-1,-1\}$	-1	-1	-1	-1	-1	-1	-1	-1	1	1	1	1	1	1	1	1
$\{-1,1\}$	-1	-1	-1	-1	1	1	1	1	-1	-1	-1	-1	1	1	1	1
$\{1,-1\}$	-1	-1	1	1	-1	-1	1	1	-1	-1	1	1	-1	-1	1	1
$\{1,1\}$	-1	1	-1	1	-1	1	-1	1	-1	1	-1	1	-1	1	-1	1

There are several inherent advantages to this numbering scheme. The first is that if a function on n variables is f_i, then the function f_i on $n+1$ variables has the same true points with a 1 prepended to each n-tuple. For example, the function f_3 on two variables is $\{[1,-1].[1,1]$, while f_3 on three variables is $\{[1,1,-1],[1,1,1]\}$. This will be useful in our discussion of Equivalence Classes below. Another advantage is that because the function numbers encode the true points, we can perform bitwise operations on the function numbers to find the result of using two subcircuits as inputs to a gate. If we have an AND gate whose inputs are circuits computing f_i and f_j, the output of the gate is f_k where k is the bitwise AND of i and j.

2.1.2 Equivalence Classes

It is often useful to think about Boolean functions in terms of equivalence classes, particularly when designing Boolean circuits to compute them. Tilman Piesk [[Pie20]] proposed Small Equivalence Classes and Big Equivalence classes as the names of particular groupings of Boolean functions.

A Small Equivalence Class (SEC) is a group of functions which can be expressed in terms of one another by negating some subset of the input variables. If $f(x_1, x_2, x_3) = g(\neg x_1, x_2, \neg x_3), \forall (x_1, x_2, x_3)$, then f and g are members of the same SEC. OEIS sequence A000231 [[OEISb]] counts the number of SEC's for n variables. We say that the *representative* of a SEC is the lowest numbered function in the SEC using the numbering scheme in table 2.1. OEIS sequence A227722 [[OEISc]] lists the first 10000 functions which are SEC representatives.

A Big Equivalence Class (BEC) is a group of functions which can be expressed as one another by negating and permuting some subset of the input variables. If $f(x_1, x_2, x_3) = g(\neg x_3, x_1, \neg x_2), \forall (x_1, x_2, x_3)$, then f and g are members of the same BEC. OEIS sequence A000616 [[OEISa]] counts the number of BEC's for n variables. We say that the representative of a BEC is the lowest numbered function in the BEC using the numbering scheme in table 2.1. OEIS sequence A227723 [[OEISd]] lists the first 10000 functions which are BEC representatives.

Any two functions in a single BEC can be computed by Boolean functions with the same number of gates. In order to permute variables, we need only replace each input to a gate by its image variable under the permutation. To negate a variable, we instead change the type of gate as discussed in section 2.1.3 below. Since we are interested primarily in finding the circuit complexity of small Boolean functions, it will suffice to find circuits which compute the representative function of each BEC. Why is this useful? Because there are $2^{2^4} = 65536$ Boolean functions of 4 variables, but there are only 402 Big Equivalence Classes. Since one of our primary objectives is to create a catalog of a minimal circuit for every Boolean function of four variables, this reduction in the size of the catalog is critical both the the efficiency of our search and the manageability of the results.

2.1.3 Straight-Line Programs

Straight-line programs are a method of implementing Boolean circuits. Each line of the program has a Boolean output associated with it, possibly dependent on the outputs of previous lines. There are no loops, if-then statements, or other control statements in a straight-line program. The input to a straight-line program is the vector $(x_1, \ldots, x_n) \in K^n$. We denote the output of line i as y_i. For an n-variable function, the first n lines are always $[1], [2], \ldots, [n]$, which sets each y_i equal to the corresponding input x_i. Program lines after the first n represent gates which take the outputs of previous lines as inputs. Table 2.2 lists the various line formats and the contexts in which those formats are used.

These programs are represented as lists of lists in Maple. The program $[[1], [2], [3], [2]]$ takes an input vector (x_1, x_2, x_3) and outputs the value of x_2, ignoring the other variables. The program $[[1], [2], [3], [1, 1, 2], [5, 3, 4]]$ outputs $x_3 \vee (x_1 \wedge x_2)$.

Gate types 1 through 10 represent the ten Boolean functions of two variables which depend on both of their inputs. The other six functions of two variables are the two constant functions, the two functions whose outputs are equal to one of the input variables, and the two functions whose outputs are the negation of one of the input variables. Gates representing these six degenerate functions are only used to

Table 2.2: Straight-line program line formats.

Line format	Function	Context
$[i]$	Sets $y_i = x_i$, where x_i is ith bit of input	Appears on line i
$[i]$	Outputs y_i $(= x_i)$	After line i (0-gate only)
$[NOT, i]$	Outputs $\neg y_i$ $(= \neg x_i)$	After line i (0-gate only)
$[TRUE]$	Outputs 1	Constant function only
$[FALSE]$	Outputs -1	Constant function only
$[1, i, j]$	Sets $y_k = y_i \wedge y_j$	On line k with $i, j < k$
$[2, i, j]$	Sets $y_k = y_i \wedge \neg y_j$	On line k with $i, j < k$
$[3, i, j]$	Sets $y_k = \neg y_i \wedge y_j$	On line k with $i, j < k$
$[4, i, j]$	Sets $y_k = y_i \oplus y_j$	On line k with $i, j < k$
$[5, i, j]$	Sets $y_k = y_i \vee y_j$	On line k with $i, j < k$
$[6, i, j]$	Sets $y_k = \neg y_i \wedge \neg y_j$	On line k with $i, j < k$
$[7, i, j]$	Sets $y_k = y_i \equiv y_j$	On line k with $i, j < k$
$[8, i, j]$	Sets $y_k = y_i \vee \neg y_j$	On line k with $i, j < k$
$[9, i, j]$	Sets $y_k = \neg y_i \vee y_j$	On line k with $i, j < k$
$[10, i, j]$	Sets $y_k = \neg y_i \vee \neg y_j$	On line k with $i, j < k$

represent zero-gate circuits and will never be used in any circuit with one or more non-degenerate gates.

As mentioned in the previous section, we need to demonstrate that the SLP representation of a circuit computing a BEC representative function can be modified to compute any other member of that BEC without changing the number of gates. If f is a BEC representative and g is some other member of that BEC, we proceed as follows. Let $\sigma : [n] \to [n]$ be a permutation of the variables and $\nu : [n] \to \{-1, 1\}$ be a function where the variables such that $\nu(x_i) = -1$ are negated. By the definition of BEC's, there exist σ, ν such that $g(x_1, \ldots, x_n) = f(\nu(1)x_{\sigma(1)}, \ldots, \nu(n)x_{\sigma(n)})$ for all input vectors $(x_1, \ldots, x_n)$.

We can now modify the circuit for f to compute g. First, we take every input i to a gate which is one of the input variables (i.e. a number in $1 \ldots n$), and replace that input with $\sigma(i)$. We now proceed systematically through every j such that

$\nu(j) = -1$. Wherever j is an input to a type 4 or 7 gate (the XOR-type gates), we switch the gate type to the other one. Wherever j is the first input to a gate of types $1, 2, 3, 5, 6, 8, 9, 10$, we change the gate to type $3, 6, 1, 9, 2, 10, 5, 8$ respectively. Wherever j is the second input to a gate of types $1, 2, 3, 5, 6, 8, 9, 10$, we change the gate to type $2, 1, 6, 8, 3, 5, 10, 9$ respectively. By making these modifications for each j in turn, we may end up switching the type of a particular gate twice. If both inputs to an XOR-type gate are negated, we will in fact end up with the original gate type. But this new circuit will compute g using the same number and structure of gates as the original circuit for f.

2.2 General Functions of Four or Fewer Variables

In this section, we describe the process through which we produce a catalog of minimal circuits for every Boolean function of up to four variables. The resultant catalogs contain a (not necessarily unique) minimal SLP for the representative function of each Big Equivalence Class. As described above, these SLP's could be modified to produce an SLP for any Boolean function in the BEC by finding the BEC representative, determining the permutation and set of negations mapping the functions to each other, and applying the above algorithm to transform the circuit.

2.2.1 Methodology

We first write procedures which create a list of the BEC representative functions of n variables. This admittedly inelegant procedure loops through function numbers $0 \ldots 2^{2^n} - 1$ using the previously discussed function numbering convention. The function is converted into our canonical format and then checked to see if it is equivalent to any previous function under signed permutation of the input variables. If not, the function is added to the list of BEC representative functions. When the process completes, the functions are written to a catalog file with each function designated "Not Found".

After the creation of the catalog, we use a procedure which generates a set of SLP's with a given number of variables and gates. We then check each SLP to see if it computes a function for which we do not have an SLP in the catalog. If it does, we associate that SLP to the function in our catalog and remove the function from the list of functions we are seeking. By running this routine for n variables and $0, 1, 2, \ldots$ gates, we eventually complete the catalog of n-variable functions.

A central issue is which SLP's to generate in such a procedure. We would like to generate all syntactically valid SLP's for n variables and g gates to guarantee that we are checking every possible circuit of a given size. But even for the relatively innocuous case of $n = 4$ and $g = 7$, there are $10^7 \prod_{i=4}^{10} i^2 \approx 3.66 \times 10^{18}$ syntactically valid SLP's. We therefore need to find a subset of those SLP's which a) is small enough that we can generate it practically, b) is complete in the sense that if there exists an SLP in the entire set for a given f there will be one in the subset, and c) we know how to produce.

The most obvious optimization is to eliminate any circuit where the output of a gate is unused. Our recursive algorithm for generating circuits takes care of that by generating the final gate and then creating one subcircuit for each input. A subcircuit can be empty, in which case it will represent either an input variable or a previously constructed gate.

The next obvious optimization is to restrict all gates to be of the form $[g, i, j]$ with $i < j$. We can safely do this because if $i = j$, the gate output is either constant (for gate types $2, 3, 4, 7, 8, 9$), the input value y_i (for gate types 1 and 5), or the negation of y_i (for gate types 6 and 10). In fact, any such gate can be eliminated, meaning that the Boolean function could be computed by a circuit with fewer gates. If $i > j$, we can reverse the inputs and change the gate type (by swapping types $2 \Leftrightarrow 3$ or $8 \Leftrightarrow 9$) if the gate is not symmetric with respect to its inputs.

A slightly less obvious simplification is to eliminate circuits and subcircuits which are mirror images of each other. The justification for doing this is that if the final gate G (which produces the overall output of the circuit) has two inputs representing subcircuits A and B, then there is another circuit whose final gate has B as the first input and A as its second which is functionally identical. Swapping the inputs may require changing the gate type of G (again by swapping types $2 \Leftrightarrow 3$ or $8 \Leftrightarrow 9$) if the gate is not symmetric with respect to its inputs. We accomplish this in our Maple code by limiting the recursive construction of circuits. When we are building a circuit with g gates, we only allow the first input to be a subcircuit of between 0 and $\lfloor (g-1)/2 \rfloor$ gates.

We realize one other (admittedly minor) optimization by restricting the inputs to any gate which has zero gates in exactly one of its subcircuits. The input corresponding to the zero-gate subcircuit is not allowed to match either input to the gate producing the other input. In this situation, which we can think of as having the child of a gate match a grandchild of that gate, we would be able to merge the two gates into one or eliminate them completely. To see why this must be the case, consider the two-gate subcircuit as a black box. The only inputs are the duplicated input and one other input. Because our gate model contains all non-degenerate functions of two variables, the black box must be equivalent to one of those functions, be constant, or be equivalent to either one of the inputs or its negation.

We were, however, somewhat overzealous with our original attempt at these reductions. In early versions of our code, any gate which had a zero-gate subcircuit as its left input was given one of the x_i as its input. While this is efficient in the sense that it provides for a very significant reduction in the number of SLP's generated, we eventually realized a flaw in this optimization. It does not allow for fan-out of gates. It is possible that the smallest circuit for computing a particular function may use the output of a gate more than once. We therefore modified the code to allow the inputs of any gate which does not have at least one gate in each of its input subcircuits to be any prior line number in the SLP (with the exception noted in the immediately preceding paragraph).

This ability to reuse prior gates dramatically increases the total number of SLP's being generated. In order to limit the impact of the change, we add one other level of pruning. As each recursive call in the chain is producing its set of SLP's, we select only one which computes the same Boolean function when considered as a SLP in its own right. This means that each level of recursion can return at most 2^{2^n} SLP's. This is the version of the code which produced the output on the author's webpage.

We note that there is a potential issue with this last pruning in the case that the optimal circuit for computing a function f uses some gate G in each subcircuit of its output gate. If there are multiple ways to compute the function represented by the left subcircuit, we may return an equivalent subcircuit which does not contain G. This issue is discussed in the following sections.

2.2.2 Results

By running our code (See Appendix B) for $1 \leq n \leq 4$ and starting at $g = 0$, we have compiled catalogs of minimal circuits for all Boolean functions of up to 4 variables. In the catalogs we produced, each entry is the representative function of a Big Equivalence Class. The first line contains the number of the representative function (as in OEIS sequence A227723 [[OEISd]]) followed by the set of true points for the function. The next line is the encoding of a minimal straight-line program which computes the function. Using the techniques discussed in sections 2.1.2 and 2.1.3 above, the circuits for a BEC representative function can be modified to compute any function in the BEC.

A summary of the circuit complexities for functions of up to 4 variables is shown in tables 2.3 and 2.4. As expected, 1-variable functions never require any gates because they are either constant or take the value or negation of the input variable (and we do not consider NOT to be a gate). Because our definition of a gate is any function of two variables which depends on both inputs, the 2-variable functions are similarly unexciting. There are 6 trivial functions ($false$, x_1, $\neg x_1$, x_2, $\neg x_2$, and $true$), and the other 10 functions are each computed by one of the gates we define.

Table 2.3: Number of BEC's by circuit complexity

vars	0-gate	1-gate	2-gate	3-gate	4-gate	5-gate	6-gate	7-gate	Total
1	3								3
2	3	3							6
3	3	3	8	5	3				22
4	3	3	8	34	59	139	130$*$	26$*$	402

The entries marked with $*$'s for four-variable functions may be inaccurate due the gate reuse issue described in the previous section. In a 2006 paper, Saarinen wrote,

Table 2.4: Number of functions by circuit complexity

vars	0-gate	1-gate	2-gate	3-gate	4-gate	5-gate	6-gate	7-gate	Total
1	4								4
2	6	10							16
3	8	30	114	80	24				256
4	10	60	456	2474	10624	24184	24784∗	2944∗	65536

“We have determined the gate complexity of all $2^{2^4} = 65536$ four-bit Boolean functions. This was done by performing an exhaustive search over all circuits with one gate, two gates, etc., until circuits for all functions had been found.” [[Saa06], p. 263] If we assume that Saarinen’s values are accurate, then the circuits we produce for several BEC’s are seven-gate circuits when some six-gate circuit would suffice. We describe a potential future project for reconciling this issue in section 2.2.3 below.

For 3 or 4 variables, a natural question is which functions have the highest circuit complexity? In the 3-variable case, there are 3 BEC’s with the maximum complexity of 4 gates. The representative functions of these classes are 22, 23, and 107. Function 22 is true when exactly two of the input variables are true. Function 23 is the majority function (at least two inputs are true). Function 107 is true when exactly one input is true OR its first two inputs are both true.

Taking the majority function as an example, the particular SLP our algorithm found to compute this function is $[[1],[2],[3],[4,1,2],[4,1,3],[1,4,5],[4,1,6]]$. In more traditional notation, this circuit is $x_1 \oplus ((x_1 \oplus x_2) \wedge (x_1 \oplus x_3))$. There are certainly other ways to compute the majority function with 4 gates, but it cannot be computed with 3 or fewer gates.

In the 4-variable case, there are 26 BEC’s which have the maximum circuit complexity of 7 gates. Many of them have a similar flavor to the 3-variable functions requiring 4 gates. A few typical examples are function 278 (exactly three inputs are true), function 279 (at least three inputs are true), and function 6015 (at least two inputs are true). Most of the other functions in this group are more akin to 107 in the 3-variable case. Function 1633 is probably best described as $((x_1 \oplus x_2) \wedge (x_3 \oplus x_4))$ or $(x_1 x_2 x_3 x_4)$.

2.2.3 Future Project

There is a potential flaw in the methodology utilized above. The discrepancy between our results and Saarinen's results for functions of four variables suggests that we have some seven-gate circuits for functions which may only require six gates. Suppose there exists some six-gate circuit to compute a function f. Further suppose that there is some gate whose output feeds (directly or indirectly) into both inputs of the final gate of this circuit. The pruning method described in the penultimate paragraph of Section 2.2.1 may inadvertently discard this circuit. If it is the only six-gate circuit which computes f, the entry in our catalog for f is not minimal.

This final pruning was added to overcome the explosive growth of the number of circuits for $n = 4$ and $g = 7$. Without this pruning, and constrained by the single-threaded and memory-intensive limitations of Maple, our existing code would take several years to execute. Fixing the issue will require a complete rewrite of the code, either in Maple or switching to a more efficient environment such as Python.

The idea for this project is the following: we modify the symmetry condition for the number of gates in the left subcircuit to allow three gates on the left and two on the right. We do this to guard against the case in which the two subcircuits use a common gate (which could be specified in either subcircuit), but each subcircuit requires three gates including the common one. Generate a list of all functions (not just Big Equivalence Classes) which can be computed using at most three gates. For each of those functions, produce and save a list of all circuits which compute the function using three or fewer gates.

Next, for every BEC representative function f which is currently assigned a seven-gate circuit in the catalog, perform the following sequence of steps:

1. Select a function g from the list of functions which can be computed in three or fewer gates. Let $\mathcal{A}$ be the set of all circuits which compute g.

2. For each gate type $t \in [1..10]$, produce a list of functions $H = \{h_1 \ldots h_k\}$ such that a t-type gate with g as its left input and h_i as its right input will output function f.

3. For each circuit $A \in \mathcal{A}$ and each $h_i \in H$, try to find a circuit B which satisfies the following conditions:

   - B computes h_i.
   - The inputs of each gate in B are chosen from input variables, the output of any gate in A, and the output of any previously constructed gate in B.
   - The total number of gates in A and B is five. This ensures that the total number of gates in the circuit is six.

4. If no such B can be constructed, select a new g and repeat the previous steps.

If the above process fails, then f has a gate-complexity of seven.

2.3 Monotone Functions of Five or Fewer Variables

There are two major advantages to restricting our analysis to monotone functions. The first is that there are many fewer monotone Boolean functions (and Big Equivalence Classes thereof) than there are general Boolean functions. The second is that we need fewer types of gates to construct Boolean circuits which compute monotone functions. In practice, the representative of each monotone Big Equivalence Class is a positive function, i.e. changing the value of an input variable from true to false will never change the output from false to true. We can therefore model the circuits for these functions using only AND and OR gates, which are gate types 1 and 5 in our existing representation.

It is important to note that the minimal circuit using only AND and OR gates may be larger than the minimal circuit if we allow the full range of gates used in the preceding section. In fact, Razborov [[Raz85]] and Tardos [[Tar88]] showed that the gap between monotone and non-monotone circuit complexities can be exponential.

2.3.1 Definitions and Notation

For the most part, we use the same conventions as we do in the general case. We do need a few definitions, which we will take directly from Crama and Hammer [[CH11]] with only notational modification.

Definition 2.2. Let f be a Boolean function on K^n, and let $k \in \{1, 2, \ldots, n\}$. We say that f is positive (respectively, negative) in the variable x_k if $f_{|x_k=-1} \leq f_{|x_k=1}$ (respectively $f_{|x_k=-1} \geq f_{|x_k=1}$). We say that f is monotone in x_k if f is either positive or negative in x_k.

Here the notation $f \leq g$ for Boolean functions f and g means $f(\overline{x}) = 1 \Rightarrow g(\overline{x}) = 1$ for all $\overline{x} \in K^n$. The notation $f_{|x_k=-1}$ is the restriction of f to input vectors where input x_k is false.

Definition 2.3. A Boolean function is positive (respectively, negative) if it is positive (respectively, negative) in each of its variables. The function is monotone if it is monotone in each of its variables.

2.3.2 Methodology

As mentioned above, a useful byproduct of the numbering scheme we are using for Boolean functions is that the lowest numbered function in each monotone BEC is a positive function. We therefore do not need to worry about negating variables in our gates, and we can use only gate types 1 and 5. This mean that while the number of circuits with each number of gates still grows exponentially, the base of the exponential is 2 rather than 10 as it is in the general case.

The other numerical advantage we have is that even with 5 variables, there are only 210 monotone BEC's, so we are hunting for many fewer needles in our proverbial haystack. By comparison, there are over 1.2 million BEC's for general functions of 5 variables, which is why we do not attempt to catalog those.

The methodology for producing this catalog is nearly identical to that for general functions. The most interesting difference (see Maple code in section B.2) is in the initialization of the catalog with the BEC representative functions. In the general case, we loop through every function and discard it if we have already seen an equivalent function under signed permutation of the variables. That is obviously not feasible for the 2^{2^5} functions of 5 variables.

Table 2.5: Number of BEC's by monotone circuit complexity

vars	0-gate	1-gate	2-gate	3-gate	4-gate	5-gate	6-gate	7-gate	Total
1	3								3
2	3	2							5
3	3	2	4		1				10
4	3	2	4	10	2	6	1	2	30
5	3	2	4	10	26	16	42	35	...
vars	8-gate	9-gate	10gate	11gate	12gate	13gate			Total
5	44	18	3	6		1			210

Instead, we start with the function $\{[1, 1, 1, 1, 1]\}$ as our working list. The main loop then pulls a function from the working list, adds it to the list of monotone functions, then calculates all of the functions which have one additional true point formed by negating one value in an existing true point. If that function is already in the main list, already in the working list, or not monotone, it is discarded. Otherwise it is appended to the working list. This allows us to generate the initial blank catalog in a reasonable amount of time.

The only other relevant difference is that the SLP generation procedure only uses AND and OR gates at every level beyond the 0-gate SLP's. See Appendix B.2 for code excerpts and additional resources.

2.3.3 Results

By running the aforementioned code for $1 \leq n \leq 5$ and starting at $g = 0$, we have compiled catalogs of minimal circuits for all monotone Boolean functions of up to 5 variables. In the catalogs we produced, each entry is the representative function of a Big Equivalence Class. The first line contains the number of the representative function (as in OEIS sequence A349743 [[OEISe]], submitted to OEIS as part of this project) followed by the set of true points for the function. The next line is the encoding of a minimal straight-line program which computes the function. A summary of the results is provided in table 2.5.

As in the general case, these SLP's could be modified to handle any function in the respective BEC. If any variables are negated in a signed permutation, so that the function is monotone without being positive, we would need to introduce gates of types other than 1 or 5. We would still not need either of the XOR-type gates, since any function which depends on the output of such a gate is not monotone (or at least could be written without such a gate).

Chapter 3

Moments of the Number of Subcubes in Boolean Functions

"The expectation functional, $\mathbb{E}[X]$ *is a powerful tool to study combinatorial objects, and often gives us quite useful information... For the higher moments,* $\mathbb{E}[X^r]$*, the computation gets complicated fast and we need computers to do symbolic computation."* - Thotsaporn Aek Thanatipanonda [[Tha20], p. 1]

This chapter overlaps significantly with a stand-alone paper on arXiv.org [[JSZ23]] which has been accepted for publication in the *Palestine Journal of Mathematics.*

3.1 Introduction

When analyzing the distribution of a combinatorial quantity or random variable X, the statistical moments of X provide information about the shape of the distribution. The first moment, denoted $\mathbb{E}[X]$, is the expected value (or mean μ). The second moment, $\mathbb{E}[X^2]$, indicates how spread out the distribution is. Higher moments yield further information about the distribution including asymmetries and the relative importance of the tails of the distribution to its shape. We will calculate both the *raw* moments and the *central* moments (often referred to as moments about the mean).

In this chapter, we consider the space of all Boolean functions on n variables. We are particularly interested in the number and sizes of implicants contained within each Boolean function. An implicant is an expression in the same variables as the function such that the function evaluates to true on any input for which the expression evaluates to true. In the traditional view of Boolean functions, we might write a function as $f(x_1, x_2) = x_1 \vee x_2$. This function has 3 implicants containing 1 point, namely x_1x_2, $x_1\overline{x_2}$, and $\overline{x_1}x_2$. It has 2 implicants with 2 points, x_1 and x_2.

Equivalently, we can think about a Boolean function as the subset of points $(x_1, \ldots, x_n) \in \{-1, 1\}^n$ for which the function evaluates to true. In this context, implicants are subcubes of the cube which are also subsets of the function. In the previous example, f would have 3 points (0-dimensional subcubes) and 2 edges (1-dimensional subcubes) contained in the 2-dimensional cube. We will primarily use the subset and subcube terminology throughout the remainder of this chapter, setting aside that our interest in these objects originally stemmed from the Boolean function interpretation.

3.1.1 Definitions and Notation

We use the same definition of a Boolean function as in 2.1.1. Let n be the number of variables, so our functions will be subsets of the n-dimensional cube. We define $\mathcal{F}_n$ to be the set of all Boolean functions on n variables. We will generally use r as the dimension of a subcube of interest. We will also use k as the number of the moment we are calculating, effectively the number of r-dimensional subcubes we are considering at a time. [1]

[1] We note that our k and r are reversed from Thanatipanonda's 2020 paper.

The combinatorial quantity we are studying should probably be denoted $X_{n,f,r}$ signifying the number of r-dimensional subcubes in the function $f \in \mathcal{F}_n$. In practice, n and f will be sufficiently obvious from context that they will be omitted for ease of notation. In addition, most of our computations will be symbolic in n. Therefore the symbol X_2 should be interpreted as the number of 2-dimensional subcubes contained in an unspecified function on n variables.

Our goal is to calculate $\mathbb{E}[X_r^k]$ and $\mathbb{E}[(X_r - \mu_r)^k]$ for as many combinations of r, k as is feasible.

3.2 Previous Results

Thanatipanonda's approach [[Tha20]] uses linearity of expectation and a variation of inclusion-exclusion. He first enumerates all of the combinations of r-subcubes of size k, arranging them by how the subcubes overlap. For example, in the case of $k = 2$ and $r = 1$, we are considering pairs of edges. Two edges can be disjoint, overlap at a point, or coincide. The total number of points included in the pair of edges is therefore 4, 3, or 2 respectively, and the probability of that particular pair of edges being present in a random subset of the n-cube is $\frac{1}{16}$, $\frac{1}{8}$, or $\frac{1}{4}$ respectively. Using inclusion-exclusion to determine the number of each such type permits a calculation of the moment (in this example $\mathbb{E}[X_1^2]$) by linearity of expectation. The calculation of the first two moments is relatively straightforward. He produces general formulas for the first two moments of r-dimensional subcubes in functions of n variables.

$$\mathbb{E}[X] = \sum_{i=1}^{\binom{n}{r}2^{n-r}} E[X_i] = \binom{n}{r} 2^{n-r} \cdot \frac{1}{2^{2^r}}$$

$$\mathbb{E}[X^2] = \sum_{i=0}^{r} \frac{\binom{n}{i,r-i,r-i,n-2r+i} 2^{n-i}}{2^{2^{r+1}}} \cdot \left(2^{2^i} - 1\right) + \frac{\left[\binom{n}{r} 2^{n-r}\right]^2}{2^{2^{r+1}}}$$

Thanatipanonda was able to use his inclusion-exclusion approach to produce formulas for the third moment for $r = 0$ and $r = 1$ but "did not manage to find formulas for other moments." [[Tha20], p. 21] The number of cases for how the subcubes can overlap grows too quickly for this method to remain practical beyond the third moment.

3.3 Methodology

For a fixed number of variables n, it is theoretically possible to generate all Boolean functions on n variables, compute the number of subcubes of each size contained in each one, and add the contributions together to find the statistical moments. This is feasible for $n = 4$, because there are only $65,536$ such functions. However, there are over 4 billion functions on 5 variables, so this naïve approach is already impractical. Furthermore, we want to produce a symbolic expression in n for each of the moments, so enumerating the functions for any particular n does not suffice.

We employ an approach which is more efficient than enumeration and less complicated than inclusion-exclusion, instead taking advantage of many available symmetries to perform direct computations on a small number of functions. We use the overlapping stages approach described by Zeilberger [[Zei04]] to build successively more efficient algorithms while ensuring the accuracy of our methods.

3.3.1 Data Structure

Every r-dimensional subcube of $\{0,1\}^n$ has the form

$$C = \{(x_1, \ldots, x_n) \in \{0,1\}^n \,|\, x_{i_1} = \alpha_{i_1}, \ldots, x_{i_{n-r}} = \alpha_{i_{n-r}}\},$$

for some $1 \leq i_1 < i_2 < \cdots < i_{n-r} \leq n$ and $(\alpha_{i_1}, \ldots \alpha_{i_{n-r}}) \in \{0,1\}^{n-r}$. A good way to represent it on a computer is as a row-vector of length n, in the *alphabet* $\{0,1,*\}$, where the entries corresponding to $i_1, i_2, \ldots, i_{n-r}$ have $\alpha_{i_1}, \ldots . \alpha_{i_{n-r}}$ respectively and the remaining r entries are filled with **wildcards**, denoted by $*$.

For example, if $n = 7$ and $r = 3$, the 3-dimensional cube

$$\{(x_1, \ldots, x_7) \in \{0,1\}^7 \,|\, x_2 = 1, x_4 = 1, x_5 = 0, x_7 = 1\},$$

is represented by

$$*1*10*1.$$

We are trying to find a weighted count of **ordered** k-tuples of r-dimensional subcubes. The natural data structure for these is the set of k by n matrices in the alphabet $\{0,1,*\}$ where every row has exactly r wildcards.

For any **specific**, numeric n, there are only $(2^{n-r}\binom{n}{r})^k$ of these matrices, and for each and every one of them one can find the cardinality of the union of the corresponding subcubes, call it v, and add $1/2^v$ to the running sum. But we want to do it for **symbolic** n, i.e. for all n. We will soon see how, for each *specific* (numeric) r and k this can be done *in principle*, but only for relatively small r and k *in practice.* An interesting consequence of our algorithm is the precise degree in n and 2^n of the expression for $\mathbb{E}[X_r^k](n)$.

3.3.2 Kernels and Remainders

A key object in our approach is the **kernel**. Given a $k \times n$ matrix M in the alphabet $\{0, 1, *\}$, we call a column **active** if it contains at least one '$*$'. Note that the matrix has exactly $k \cdot r$ '$*$'s, hence the number of **active columns**, call it a, is between r and $k \cdot r$. [2] The **kernel** of M is the submatrix of M consisting of its active columns.

We will say that a matrix is in **canonical form** if the active columns are occupied by the a leftmost columns (i.e. its kernel is contiguous starting in the first column). Obviously, there are $\binom{n}{a}$ ways to choose which of the n columns are active, therefore we can compute the contribution to the expectation for the set of matrices in canonical form and multiply by $\binom{n}{a}$.

It remains to do a *weighted-count* in which every matrix contributes $1/2^v$, where v is the cardinality of the union of the corresponding subcubes represented by the k rows, for the set of matrices in canonical form. Note that there are only **finitely many** choices for the a leftmost columns, i.e. the set of $k \times a$ matrices in the alphabet $\{0, 1, *\}$ with the property that every column has at least one '$*$', and every row has exactly r '$*$'s. These can be divided into *equivalence classes* obtained by permuting rows and columns and transposing 0 and 1 in any given column. Once these are sorted into equivalence classes, one needs only examine one representative, and then multiply the weight by the cardinality of the class.

But what about the $n - a$ rightmost columns? We refer to these submatrices as *remainders*. There are $2^{k(n-a)}$ possible submatrices; there are no wildcards in this region, so the alphabet here is $\{0, 1\}$. Almost all of these have distinct rows, more

[2] More generally, if we want to find an expression for the *mixed moment* $\mathbb{E}[X_{r_1} \cdots X_{r_k}]$ the number of active columns is between $\max(r_1, \ldots, r_k)$ and $r_1 + \cdots + r_k$.

precisely,

$$\binom{2^{n-a}}{k} k!$$

of them, and these will produce the smallest possible weight in conjunction with any kernel. The other extreme is that all the rows of the remainder are identical, and then there are only 2^{n-a} choices to fill them in.

In general, every remainder determines a **set partition** of the set of rows $\{1, \ldots, k\}$, where two rows are *roommates* if they have the same last $n-a$ entries. If that set partition has m members $1 \leq m \leq k$, then the number of choices of assigning **different** $\{0,1\}$ vectors of length $n-a$ to each of the parts of the set partition is

$$\binom{2^{n-a}}{m} m!.$$

Now for each a and for each set partition, we let the computer generate the **finite** set of $k \times a$ matrices in the alphabet $\{0, 1, *\}$. Each of the members of the set partition has its own submatrix, and we ask our computer to find the number of vertices in the corresponding union of subcubes corresponding to each member of the examined set partition. Since they are disjoint, we add them up, getting v for that particular pair (matrix, set partition), giving credit $1/2^v$.

3.4 Moments of Numbers of 1-dimensional Subcubes

Our computations confirm the first three moments for 1-dimensional subcubes as computed by Thanatipanonda [[Tha20]]. Our approach produces the fourth, fifth, and sixth moments as well.

Fourth (raw) moment for edges:

$$\mathbb{E}[X_1^4] = \frac{1}{4096}(n^4 2^{4n} + 24n^4 2^{3n} + 144n^4 2^{2n} + 160n^4 2^n + 12n^3 2^{3n} + 48n^3 2^{2n} - 192n^3 2^n + 12n^2 2^{2n} + 48n^2 2^n - 64n 2^n)$$

Fourth central moment for edges:

$$\mathbb{E}[(X_1 - \mu_1)^4] = \frac{1}{1024}(40n^4 2^n - 48n^3 2^n + 12n^2 2^n - 16n 2^n + 12n^4 2^{2n} + 12n^3 2^{2n} + 3n^2 2^{2n})$$

Fifth (raw) moment for edges:

$$\mathbb{E}[X_1^5] = \frac{n^2 2^n}{32768}(n^3 2^{4n} + 40n^3 2^{3n} + 480n^3 2^{2n} + 1760n^3 2^n + 640n^3 + 20n^2 2^{3n} + 240n^2 2^{2n} - 480n^2 2^n - 3840n^2 + 60n 2^{2n} + 240n 2^n + 1280n - 320 \cdot 2^n)$$

Fifth central moment for edges:

$$\mathbb{E}[(X_1 - \mu_1)^5] = \frac{5n^3 2^n}{1024}\left(6n^2 2^n + 4n^2 + 3n 2^n - 24n + 8\right)$$

Sixth (raw) moment for edges:

$$\mathbb{E}[X_1^6] = \frac{n 2^n}{262144}(n^5 2^{5n} + 60n^5 2^{4n} + 1200n^5 2^{3n} + 9120n^5 2^{2n} + 19200n^5 2^n - 14336n^5 + 30n^4 2^{4n} + 720n^4 2^{3n} + 1440n^4 2^{2n} - 29760n^4 2^n - 42240n^4 + 180n^3 2^{3n} + 1440n^3 2^{2n} + 4800n^3 2^n + 30720n^3 - 840n^2 2^{2n} - 2400n^2 2^n + 30720n^2 - 1920n 2^n - 53760n + 38912)$$

Sixth central moment for edges:

$$\begin{aligned}&\mathbb{E}[(X_1-\mu_1)^6] = \\ &\frac{n2^n}{32768}\cdot\big(120n^5 2^{2n} + 1920n^5 2^n - 1792n^5 - 840n^4 2^n + 180n^4 2^{2n} - 5280n^4 \\ &\qquad + 90n^3 2^{2n} - 360n^3 2^n + 3840n^3 + 15n^2 2^{2n} - 300n^2 2^n + 3840n^2 \\ &\qquad - 240n2^n - 6720n + 4864\big)\end{aligned}$$

From these moments, it follows that (as n approaches infinity) the third through sixth scaled moments about the mean converge to 0, 3, 0, and 15 respectively. This suggests that the random variable X_1 is asymptotically normal. Konieczna [[Kon93]] proved the asymptotic normality of X_r in the more general case where each subcube appears with probability $p \in (0, 1)$.

3.5 Moments of Numbers of 2-dimensional Subcubes

Our computations confirm the first two moments for 2-dimensional subcubes as computed by Thanatipanonda [[Tha20]]. We are able to find the third and fourth moments as well.

Third (raw) moment for squares:

$$\begin{aligned}\mathbb{E}[X_2^3] = \frac{n(n-1)2^n}{2097152}&(n^4 2^{2n} + 48n^4 2^n + 576n^4 - 2n^3 2^{2n} + 384n^3 + n^2 2^{2n} \\ &+ 24n^2 2^n + 1344n^2 - 72n2^n - 1024n - 2176)\end{aligned}$$

Third central moment for squares:

$$\mathbb{E}[(X_2-\mu_2)^3] = \frac{n\,(n-1)\,2^n}{32768}\left(9n^4+6n^3+21n^2-16n-34\right)$$

Fourth (raw) moment for squares:

$$\begin{aligned}&\mathbb{E}[X_2^4] = \\ &\frac{n(n-1)2^n}{268435456}(n^6 2^{3n} + 96n^6 2^{2n} + 3072n^6 2^n + 33280n^6 - 3n^5 2^{3n} - 96n^5 2^{2n} \\ &\qquad - 1536n^5 + 3n^4 2^{3n} + 48n^4 2^{2n} + 5376n^4 2^n + 81408n^4 - n^3 2^{3n} \\ &\qquad - 192n^3 2^{2n} - 10240n^3 2^n - 53760n^3 + 144n^2 2^{2n} - 5184n^2 2^n \\ &\qquad - 334848n^2 + 6976n 2^n - 177152n + 15360)\end{aligned}$$

Fourth central moment for squares:

$$\begin{aligned}&\mathbb{E}[(X_2-\mu_2)^4] = \\ &\frac{n\,(n-1)\,2^n}{4194304}\cdot\big(12n^6 2^n + 520n^6 + 12n^5 2^n - 24n^5 + 24n^4 2^n + 1272n^4 - 12n^3 2^n \\ &\qquad - 840n^3 - 9n^2 2^n - 27n 2^n - 5232n^2 - 2768n + 240\big)\end{aligned}$$

3.6 Moments of Numbers of 3-dimensional Subcubes

Our computations confirm the first two moments for 3-dimensional subcubes as computed by Thanatipanonda [Tha20]. We are able to find the third moment as well.

Third (raw) moment for cubes:

$$\begin{aligned}\mathbb{E}[X_3^3] = \frac{n(n-1)(n-2)2^n}{1855425871872}(&n^6 2^{2n} + 192n^6 2^n + 10752n^6 - 6n^5 2^{2n} - 288n^5 2^n \\ &+ 18432n^5 + 13n^4 2^{2n} + 3360n^4 2^n + 367872n^4 \\ &- 12n^3 2^{2n} + 6480n^3 2^n + 1571328n^3 \\ &+ 4n^2 2^{2n} - 44592n^2 2^n + 5206272n^2 \\ &+ 34848n2^n + 11860992n - 17750016)\end{aligned}$$

Third central moment for cubes:

$$\begin{aligned}\mathbb{E}[(X_3 - \mu_3)^3] = \frac{n(n-1)(n-2)2^n}{2415919104}(&14n^6 + 24n^5 + 479n^4 + 2046n^3 \\ &+ 6779n^2 + 15444 - 23112)\end{aligned}$$

3.7 Mixed Moments

We also compute a number of mixed moments, such as $\mathbb{E}[X_1 X_2]$. These moments are of interest because they provide insight into the correlation between the presence of subcubes of different sizes.

The first method for producing these mixed moments is the brute force enumeration of functions. As in previous cases, this is only feasible up to $n = 4$. Using the `Moms` routine described above, we can generate mixed moments up to 10-dimensional subcubes. The first six mixed moments are

$$\begin{aligned}
\mathbb{E}[X_0X_1] =& \frac{n2^n}{16}(2^n+2)\\
\mathbb{E}[X_0X_2] =& \frac{n(n-1)2^n}{256}(2^n+4)\\
\mathbb{E}[X_0X_3] =& \frac{n(n-1)(n-2)2^n}{24576}(2^n+8)\\
\mathbb{E}[X_1X_2] =& \frac{n(n-1)2^n}{1024}(n2^n+8n+8)\\
\mathbb{E}[X_1X_3] =& \frac{n(n-1)(n-2)2^n}{98304}(n2^n+16n+24)\\
\mathbb{E}[X_2X_3] =& \frac{n(n-1)(n-2)2^n}{1572864}(n^22^n-n2^n+32n^2+64n+240)
\end{aligned}$$

We generalize Thanatipanonda's formula for second moments to the case of the mixed moment $\mathbb{E}[X_rX_s]$ as follows:

$$\begin{aligned}
\mathbb{E}[X_rX_s] &= \sum_{i=0}^{\min(r,s)} 2^{r+s-2i}\binom{n}{i,r-i,s-i,n-r-s+i}\frac{2^{n-r-s+i}}{2^{2^r+2^s-2^i}}+\frac{Rest}{2^{2^r+2^s}}\\
&= \sum_{i=0}^{\min(r,s)} \frac{\binom{n}{i,r-i,s-i,n-r-s+i}2^{n-i}}{2^{2^r+2^s}}\cdot\left(2^{2^i}-1\right)+\frac{\binom{n}{r}2^{n-r}\binom{n}{s}2^{n-s}}{2^{2^r+2^s}}
\end{aligned}$$

where $Rest = \binom{n}{r}2^{n-r}\binom{n}{s}2^{n-s} - \sum_{i=0}^{\min(r,s)}\binom{n}{i,r-i,s-i,n-r-s+i}2^{n-i}$.

As suggested in the original paper [[Tha20], p. 22], we can think of the i in the summation as the dimension of the intersection between two subcubes. This intersection can be no larger than $\min(r,s)$. The multinomial coefficient represents the number of ways to select the i columns with wildcards in both rows, $r-i$ and $s-i$ columns with a wildcard in one row but not the other, and $n-r-s+i$ columns with no wildcards.

This generalized formula computes mixed moments more efficiently than the `Moms` procedure. We are now able to compute mixed moments up to $\mathbb{E}[X_{19}X_{20}]$.

3.7.1 Correlation

Recall that the correlation of two random variables X, Y is:

$$\mathbf{Cor}(X,Y) = \frac{\mathbb{E}[XY] - \mathbb{E}[X]\mathbb{E}[Y]}{\sqrt{\left(\mathbb{E}[X^2] - \mathbb{E}[X]^2\right)\left(\mathbb{E}[Y^2] - \mathbb{E}[Y]^2\right)}}$$

Now, having the ability to calculate mixed moments lets us find correlation. It certainly seems reasonable that the correlation between any two sizes of subcube should approach 1 as $n \to \infty$, since functions with more edges will also have more squares, etc. It seems natural to wonder how quickly the correlations approach 1. Here are some sample asymptotics:

$$\begin{aligned}
\mathbf{Cor}(X_0, X_1) &= 1 - \frac{1}{4n} + \frac{3}{32\,n^2} - \frac{5}{128\,n^3} + \frac{35}{2048\,n^4} - \frac{63}{8192\,n^5} + O\left(\frac{1}{n^6}\right) \\
\mathbf{Cor}(X_0, X_2) &= 1 - \frac{1}{n} - \frac{1}{4n^2} + \frac{1}{n^3} - \frac{21}{32\,n^4} - \frac{21}{32\,n^5} + O\left(\frac{1}{n^6}\right) \\
\mathbf{Cor}(X_0, X_3) &= 1 - \frac{9}{4n} - \frac{261}{32\,n^2} - \frac{1317}{128\,n^3} + \frac{508275}{2048\,n^4} + \frac{3004953}{8192\,n^5} + O\left(\frac{1}{n^6}\right) \\
\mathbf{Cor}(X_0, X_4) &= 1 - \frac{4}{n} - \frac{39}{n^2} - \frac{625}{n^3} - \frac{148811}{4\,n^4} + \frac{697875}{2\,n^5} + O\left(\frac{1}{n^6}\right) \\
\mathbf{Cor}(X_1, X_2) &= 1 - \frac{1}{4n} - \frac{37}{32n^2} + \frac{131}{128n^3} + \frac{115}{2048n^4} - \frac{10543}{8192n^5} + O\left(\frac{1}{n^6}\right) \\
\mathbf{Cor}(X_1, X_3) &= 1 - \frac{1}{n} - \frac{45}{4n^2} - \frac{79}{4n^3} + \frac{7595}{32n^4} - \frac{21735}{32n^5} + O\left(\frac{1}{n^6}\right)
\end{aligned}$$

$$\mathbf{Cor}(X_1,X_4) = 1 - \frac{9}{4n} - \frac{1485}{32n^2} - \frac{88509}{128n^3} - \frac{78400125}{2048n^4} + \frac{2327192169}{8192n^5} + O\left(\frac{1}{n^6}\right)$$
$$\mathbf{Cor}(X_1,X_5) = 1 - \frac{4}{n} - \frac{131}{n^2} - \frac{5000}{n^3} - \frac{4357195}{4n^4} - \frac{8037710954}{n^5} + O\left(\frac{1}{n^6}\right)$$
$$\mathbf{Cor}(X_2,X_3) = 1 - \frac{1}{4n} - \frac{173}{32\,n^2} - \frac{5461}{128\,n^3} + \frac{343299}{2048\,n^4} + \frac{6491553}{8192\,n^5} + O\left(\frac{1}{n^6}\right)$$
$$\mathbf{Cor}(X_2,X_4) = 1 - \frac{1}{n} - \frac{137}{4\,n^2} - \frac{809}{n^3} - \frac{1271453}{32\,n^4} + \frac{7260179}{32\,n^5} + O\left(\frac{1}{n^6}\right)$$
$$\mathbf{Cor}(X_2,X_5) = 1 - \frac{9}{4n} - \frac{3573}{32\,n^2} - \frac{690357}{128\,n^3} - \frac{2252335149}{2048\,n^4} - \frac{65861548283271}{8192\,n^5} + O\left(\frac{1}{n^6}\right)$$

It is obvious that the coefficient on $\frac{1}{n}$ must be negative, since the correlation can never exceed 1. More specifically, the coefficient of $\frac{1}{n}$ in $\mathbf{Cor}(X_r, X_s)$ is always $\frac{(r-s)^2}{4}$. The coefficients of lower order terms are more complicated, so we find these by computing $\mathbf{Cor}(X_r, X_s)$ for as many particular pairs (r, s) as possible and analyzing the pattern of the coefficients. For each $\frac{1}{n^i}$, we treat the matrix of coefficients of that term in $\mathbf{Cor}(X_r, X_s)$ as a function of r and s. We then attempt to find a polynomial which describes that function whose degree in r (respectively s) is at least two less than the number of rows (respectively columns) of the coefficient matrix. We find the following polynomials for the first four nonconstant terms:

$$\begin{aligned}
\frac{1}{n} &: \quad -\frac{(r-s)^2}{4} \\
\frac{1}{n^2} &: \quad -\frac{(r-s)^2}{32}\left(11r^2 + 26rs + 11s^2 - 28r - 28s + 14\right)
\end{aligned}$$

$$\begin{aligned}
\frac{1}{n^3}: \quad & -\frac{(r-s)^2}{384}\big(661r^4+1388r^3s+2166r^2s^2+1388rs^3+661s^4\\
&-4380r^3-9012r^2s-9012rs^2-4380s^3\\
&+10078r^2+17236rs+10078s^2\\
&-9600r-9600s+3256\big)\\
\frac{1}{n^4}: \quad & -\frac{(r-s)^2}{6144}\big(500327r^6+1005942r^5s+1510761r^4s^2+2021492r^3s^3\\
&\quad+1510761r^2s^4+1005942rs^5+500327s^6\\
&-6062280r^5-12161448r^4s-18268176r^3s^2\\
&\quad-18268176r^2s^3-12161448rs^4-6062280s^5\\
&+29496884r^4+59093296r^3s+77488056r^2s^2\\
&\quad+59093296rs^3+29496884s^4\\
&-73568880r^3-141137616r^2s-141137616rs^2-73568880s^3\\
&+98912180r^2+160996184rs+98912180s^2\\
&-67796448r-67796448s+18518112)
\end{aligned}$$

Coefficients for additional terms are available in Appendix C. We observe that for each $\frac{1}{n^i}$, the degree of the corresponding polynomial for its coefficients is $2i$. Each polynomial is uniformly 0 when $r = s$, which we expect because $\mathbf{Cor}(X_r, X_r) = 1$ by definition. We also note that these polynomials are symmetric in r, s, which must be true because $\mathbf{Cor}(X_r, X_s) = \mathbf{Cor}(X_s, X_r)$.

Chapter 4

Combinatorial Game - Juniper Green

"Consider any game played with a number of heaps in which each move affects just one of the heaps on the table, and in which exactly the same moves are available to each player. Any position in such a game is therefore the sum of its single heap positions, so the game is solved when we know the value of a heap of n beans for every n. Moreover, since the games are impartial, each such value is a Nim-heap, $*m$." Berlekamp, Conway, and Guy [[BCG01], p. 82]

4.1 Introduction

Games have existed since pre-historic times, dating back at least as far as the first time one of our bipedal ancestors realized that when one throws the knuckle bones of last night's dinner, those bones can land in various positions. Mathematicians have applied their techniques to the analysis of games for hundreds of years, since Cardano, Pascal, and others applied their reasoning to popular games of chance. Game theory became its own branch of mathematics about a century ago when von Neumann analyzed perfect-information zero-sum games. In the 1930's, Sprague (in

Germany) and Grundy (in England) independently derived what is now known as the Sprague-Grundy Theorem, launching the field of combinatorial game theory.

4.1.1 Combinatorial Games - Definitions

A *combinatorial game* is an open-information game without randomness (in the form of dice or similar devices) in which any current position of the game leads to a finite number of new positions via a legal move by the player whose turn it is. Each player has "perfect information" in the sense that they are aware of the current state of the game and know what moves are available to every player. We will generally refer to the current state of the game as the *position* of the game. Frequently, the goal of the game is to be the last player to make a legal move.

A combinatorial game is *finite* if it is guaranteed to produce a winner after some finite sequence of moves. In particular, this means that there can not be any cyclical sequence of moves (i.e. a situation in which the position of the game is identical before and after a non-zero number of moves). In some sense, this is equivalent to saying that the space of eventually reachable positions of the game is strictly decreasing. Tic-tac-toe is a finite combinatorial game because each move permanently uses one square on the board, so the total number of moves to reach a conclusion cannot exceed the number of spaces on the board. Checkers is not a finite game; if each player has promoted at least one checker to a king, those kings can move between spaces in a manner which repeats a previous position.

A combinatorial game is called *impartial* if any move that is legal for one player is also legal for the other. For example, chess is a combinatorial game by these definitions, but it is not impartial. All of the pieces are visible, and the possible moves for each piece depend only on the current position, so the perfect information requirement is met. The first player can only move the white pieces, and the second the black, so the same moves are not legal for both of them.

Nim is perhaps the prototypical impartial combinatorial game. In this game, there are a finite number of objects divided into some number of distinct piles. Each player in turn removes one or more objects from a single pile. The player who cannot make a legal move (because all objects have been removed) loses the game. There are many variations of Nim. In the misère version, the player who removes the last piece loses. There are also variations in which there is a maximum number of objects which can be removed, players can remove objects from more than one pile in a turn, or there is a preliminary phase in which the object are placed into piles before the game starts.

4.1.2 Position Graphs

We can model a finite combinatorial game as a directed graph. The vertices represent positions, and there is an edge from position A to position B if and only if there is a legal move from position A which results in position B. There is one distinguished vertex representing the initial position of the game (e.g. the empty tic-tac-toe board). There can be no cycles if the game is finite, but the underlying undirected graph need not be a tree. It may be possible for multiple sequences of moves to arrive at identical positions.

We will use the notation $\mathcal{N}(A) = \{B_1, B_2, \ldots, B_k\}$ to indicate that from position A, there are legal moves resulting in positions $B_1 \ldots B_k$. If $\mathcal{N}(A) = \emptyset$, there are no legal moves from that position - the player whose turn it is to move has lost the game.

In fact, we can classify every position as either a P-position (the player who made the previous move achieving this position will win the game) or an N-position (the player about to make the next move from this position will win the game). These statements about who will win do rely on both players choosing rationally among their available moves. The algorithm for classifying all positions as N or P from the directed graph is as follows:

1. Label every position V such that $\mathcal{N}(V) = \emptyset$ as a P-position because the next player has no legal move and the previous player has won.

2. For every unlabeled position W, check whether any position in $\mathcal{N}(W)$ is already labeled as a P-position. If so, label W as an N-position because the player about to move can choose to go to a P-position and win.

3. For every unlabeled position X, check whether every position in $\mathcal{N}(X)$ is already labeled as an N-position. If so, label X as a P-position because every possible move for the current player gives the opponent a winning strategy.

4. If any position remains unlabeled, repeat steps 2 and 3.

This process must terminate for the position graph of a finite game. There are no cycles, so any sequence of moves through the graph must be of finite length. If there are no edges leaving a vertex, that position will be classified in step 1. The first iteration of steps 2 and 3 will certainly classify every vertex from which the maximum path length is 1 (and some whose maximum path length is longer). Each subsequent iteration will classify all previously unclassified vertices whose maximum path length is the number of the iteration. Therefore, the process must terminate after no more than d iterations of steps 2 and 3, where d is the maximum length of a path starting at the distinguished vertex representing the initial position.

4.1.3 Sprague-Grundy Values

We define the *minimum excluded number* of a set A, denoted $\mathbf{mex}(A)$, as the smallest non-negative integer which is not an element of A. Some examples:

$$\mathbf{mex}(\{0,1,2,3,5,6,7\}) = 4$$
$$\mathbf{mex}(\{1,2,3\}) = 0$$
$$\mathbf{mex}(\emptyset) = 0$$

The Sprague-Grundy value of a position V (often called the Grundy value) is defined recursively as follows:

$$\mathcal{G}(V) = \begin{cases} 0 & \text{if } \mathcal{N}(V) = \emptyset \\ \mathbf{mex}(\mathcal{N}(V)) & \text{otherwise} \end{cases}$$

The Sprague-Grundy value of a position V is 0 if and only if V is a P-position by the definitions of the previous section. The reason for this is that the two ways that $\mathcal{G}(V)$ can be assigned the value 0 are because there are no legal moves or because $\mathbf{mex}(\mathcal{N}(V)) = 0$. In the former case, there is no legal move for the current player. In the latter case, every legal move has a nonzero Sprague-Grundy value, so there are no options for the current player to force their opponent to lose. Therefore $\mathcal{G}(V)$ includes the information of whether V is a P-position or an N-position.

Sprague-Grundy values also provide deeper information. Perhaps the simplest way to understand this information is in terms of Nim. A game of Nim with one pile is trivial: the player whose turn it is can remove all of the objects in the pile to win, unless the pile is already empty. It is clear from the recursive definition above that a position in a single-pile Nim game has a Sprague-Grundy value equal to the number of objects. It may seem from this observation that the Sprague-Grundy value is merely a measure of how many fundamentally different mistakes the current player can make, and for the single-pile game that may well be the case.

For Nim with more than one pile, the power of the Sprague-Grundy value becomes apparent. We denote a position in k-pile Nim as $X = (x_1, x_2, \ldots, x_k)$ where x_i is the number of objects remaining in pile i. It is well-known that $\mathcal{G}(X) = \bigoplus_{i=1}^{k} x_i$, i.e. the Sprague-Grundy value of the position is the bitwise XOR of the pile sizes. [[BCG01]]

4.2 Juniper Green

Juniper Green is a two-player impartial combinatorial game invented by teacher Richard Porteous [[Ste97], [SB99]]. The basic game uses a board labeled with the integers 1 to n, with a typical value of $n = 100$. The first player chooses any number on the board. The second player then chooses any integer in $[1, n]$ which is either a factor or an integer multiple of the first. The players alternate selecting a factor or multiple of the current number, but they may not choose any number which has been selected previously. Whichever player is left without a legal move loses.

The basic game (which we refer to as "easy" in the accompanying Maple package) is trivial for any $n \geq 7$ except for $n = 10$. The first player can select any prime number $p > \lfloor \frac{n}{2} \rfloor$. The second player must choose 1, since it is the only unselected factor of p and all multiples are greater than n. The first player then chooses another prime greater than $\lfloor \frac{n}{2} \rfloor$ and the second player has no response. The case $n = 10$ is exceptional because there is only one prime in the interval $[6, 10]$, so the strategy fails.

The standard version of the game ("hard" in the Maple package), and the one that is typically played, adds one additional rule: the first player must select an even number. When considering strategy, it is still in a player's best interest to force their opponent to select 1. As soon as either player does so, their opponent can select a large prime and the game ends.

4.2.1 Winning Strategies

For the standard game, Julien Lemoine [[Lem22]] has determined whether the first player has a winning strategy for all values of n. He models the game as an undirected graph in which each integer from 1 to n is represented by a vertex, and two vertices are connected if and only if the larger number is a multiple of the smaller number. (Note that this is not a position graph as defined in Section 4.1.2.) When a player selects a number, that vertex is highlighted as the current vertex. When the next number is selected, the highlighted vertex and its associated edges are deleted from the graph. This method simplifies the analysis because once the graph becomes disconnected, one need only consider the component containing the active vertex. A further simplification results from the fact stated above that the player who chooses 1 loses coupled with the rule that the initial choice must be even: we can delete the vertex 1 and all of the vertices for primes greater than $\frac{n}{2}$, since these are now isolated vertices not available for initial selection.

One striking example of the power of this model is the case where there are at least three primes p, q, r such that $\frac{n}{4} < p < q < r \leq \frac{n}{3}$. With 1 removed, vertex p is adjacent only to $2p$ and $3p$ and similarly for q and r. In turn, $2p$ is adjacent to only 2 and p, and $3p$ is adjacent only to 3 and p. This results in the configuration shown in Figure 4.1.

For any game including such a subgraph, the first player can win by selecting any

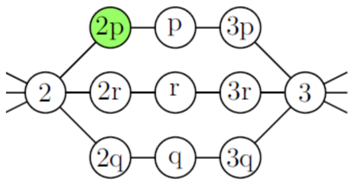

Figure 4.1: The subgraph containing primes p, q, r

of the pictured vertices adjacent to 2 or 3. Say the first player chooses $2p$ as indicated in the diagram. The second player can only choose to move left or right, eventually selecting either 2 or 3. Then the first player can select the adjacent vertex in the center row, eventually forcing the second player to select the other of 2 and 3. Now the first player selects the adjacent vertex in the bottom row and must win two moves later.

4.3 Methodology

Our initial implementation includes three procedures for each variation. The first of these generates a (potentially empty) set of positions corresponding to the legal moves from the current position. The second procedure generates a (potentially empty) set of positions for the winning moves from the current position. The final procedure calculates the Sprague-Grundy value of a given position. These procedures naïvely compute legal moves by considering the currently selected number and finding all unused legal moves, as a human player presumably would. The Sprague-Grundy value is recursively computed by generating the list of moves, finding the Sprague-Grundy value of each of the resultant positions, and taking the **mex** of that set of values.

We achieve a slightly greater efficiency with a modified set of procedures. In this second version, the legal moves are not recomputed at each step of the game. Instead, we generate a directed graph (similar to the graph used by Lemoine) with vertices representing the numbers 1 through n. The graph contains an edge from i to j if and only if j is a legal move from i, supposing that j has not previously been selected. The graph is stored as a Maple table whose indices are the vertices and whose entries are the set of vertices reachable from the index under the rules of the current variation in the initial position. We will refer to this structure as the adjacency table below. Using this stored data structure allows us to compute the moves from any position by subtracting the set of previously selected numbers from the entry in this table for the current selection without recomputing the legal moves. We use a directed graph because while the original game is symmetric (i.e. if $i \to j$ is a legal move, so is $j \to i$, several of the variations are not.

Details of the implementations for two of the variations are included in the following sections. Links to the code for all variations are available in Appendix D.

4.3.1 Traditional Game - "Hard Version"

For the initial implementation, each procedure takes two arguments: the number of spaces on the board n and the current position P. The position is a two-element list $[c, S]$ consisting of the currently selected number and a set of numbers which have already been selected. We use the convention that the initial position (player one is about to make their first selection) is represented as $[n, \{\}]$.

The procedure for computing the list of legal moves from a position is `Mh(n,P)`. If the second argument is $[n, \{\}]$, the procedure returns all positions for which the current and only selection is an even number. Otherwise, it computes all factors and multiples of c which are in the interval $[1..n]$, removes all elements of S from that list, then returns the set of positions corresponding to each of the remaining options being chosen from the current position. The following sequence of calls to `Mh` represent a possible game played with $n = 10$:

- `Mh(10,[10,{}])` returns $\{[2,\{2\}],[4,\{4\}],[6,\{6\}],[8,\{8\}],[10,\{10\}]\}$, because player one must initially select an even number.

- `Mh(10,[4,{4}])` returns $\{[1,\{1,4\}],[2,\{2,4\}],[8,\{4,8\}]\}$. After player one has selected 4, player two must select a factor or multiple of 4.

- `Mh(10,[1,{1,4}])` returns $\{[2,\{1,2,4\}],[3,\{1,3,4\}],[5,\{1,4,5\}],[6,\{1,4,6\}],$ $[7,\{1,4,7\}],[8,\{1,4,8\}],[9,\{1,4,9\}],[10,\{1,4,10\}]\}$. Player two has made a terrible mistake by selecting 1! Player one can now choose any unused number.

- `Mh(10,[7,{1,4,7}])` returns $\{\}$. Player one has chosen 7, and player two has no legal move. Player one wins the game.

The procedure for computing the list of winning moves from the current position is `WMh(n,P)`. This procedure is not used in the computation of Sprague-Grundy values, but it provides a way to determine the correct sequence of move for a player who is in a winning position. Some examples from the game traced in the previous example:

- `WMh(10,[10,{}])` returns $\{\}$. Player one does not have an initial selection from which they can win the game without an error by player two.

- `WMh(10,[1,{1,4}])` returns $\{[6,\{1,4,6\}],[7,\{1,4,7\}],[8,\{1,4,8\}],[9,\{1,4,9\}]\}$. Player one had several winning options at this point, although 7 was the only choice which resulted in an immediate win.

The procedure for computing the Sprague-Grundy value of a position is `Gh(n,P)`. If P is in the form $[n,\{\}]$, this procedure computes the **mex** of the positions representing an initial selection of an even number. Otherwise, it computes the **mex** of $Gh(n,P_i)$ for all $P_i \in Mh(n,P)$. Continuing with the above example:

- `Gh(10,[10,{}])` returns 0. Player one does not have an initial selection from which they can win the game without an error by player two, so this position is a P-position.

Table 4.1: Output of `Mhgraph(10)`

Index	Entry
1	$\{2,3,4,5,6,7,8,9,10\}$
2	$\{1,4,6,8,10\}$
3	$\{1,6,9\}$
4	$\{1,2,8\}$
5	$\{1,10\}$
6	$\{1,2,3\}$
7	$\{1\}$
8	$\{1,2,4\}$
9	$\{1,3\}$
10	$\{1,2,5\}$

- `Gh(10,[1,{1,4}])` returns 3. This value is positive, so this position is an N-position. Player one has winning options for which the SG value of the ensuing position is 0 (selecting 6, 7, 8, or 9) and losing options where the SG value of the ensuing position is 1 (selecting 3 or 5) or 2 (selecting 2 or 10).

The improved implementation is principally designed to produce Sprague-Grundy values. There is no procedure for computing winning moves, although one could use the existing procedures to find moves with positive Sprague-Grundy values to accomplish that task. This implementation does not recompute the factors and multiples of c at every stage. Instead, the procedure `Mhgraph(n)` produces and stores an adjacency table for the game. See Table 4.1 for an example.

The procedure `Ghgraph1(n,c,S)` computes the Sprague-Grundy value for a position in the game with n numbers, current selection c, and set of numbers already selected S. It first produces a (possible empty) list of legal moves by subtracting S from the entry in the adjacency table corresponding to c. If there are no legal moves, the procedure returns 0. If there are legal moves, we return the **mex** of the Sprague-Grundy values of those moves from the current position.

The procedure `Ghgraph(n)` returns the Sprague-Grundy value of the initial position for a board with n spaces and player one about to make the initial selection. It first calls `Mhgraph` to produce the adjacency graph, then finds the **mex** of the Sprague-Grundy values of every even initial selection c using `Ghgraph1(n,c,{c})`.

4.3.2 Additive Version

In the additive variation, described in more detail in Section 4.6 below, the parameters of a game are integer n and sets A, B. The number of spaces on the board is still n, but the legal moves are no longer factors and multiples of the current selection. Instead, players can either subtract an element of A or add an element of B to the current number.

In the initial implementation for this variation, each procedure takes four arguments: the number of spaces on the board$n$, the current position P, and the sets A and B. The position is again a two-element list $[c, S]$ consisting of the currently selected number and a set of numbers which have already been selected. We retain the convention that the initial position (player one is about to make their first selection) is represented as $[n, \{\}]$.

The procedure for computing the list of legal moves from a position is `Ma(n,P,A,B)`. If the second argument is $[n, \{\}]$, the procedure returns all positions for which the current and only selection is any number in $[1..n]$. Otherwise, it makes a list of all numbers between 1 and n of the form $c - a,\ a \in A$ or $c + b,\ b \in B$, removes all elements of S from that list, then returns the set of positions corresponding to each of the remaining options being chosen from the current position. The following sequence of calls to `Ma` represent a possible game played with $n = 10,\ A = B = \{1, 2\}$:

- `Ma(10,[10,{}],{1,2},{1,2})` returns the set containing $[i, \{i\}]$ for every $1 \leq$

$i \leq 10$

- `Ma(10,[3,{3}],{1,2},{1,2})` returns $\{[1,\{1,3\}],[2,\{2,3\}],[4,\{3,4\}],[5,\{3,5\}]\}$. After player one has selected 3, player two must select 1, 2, 4, or 5.

- `Ma(10,[2,{2,3}],{1,2},{1,2})` returns $\{[1,\{1,2,3\}],[4,\{2,3,4\}]\}$. Player two has made a terrible mistake by selecting 2! Player one can now choose 1 and win the game.

- `Ma(10,[1,{1,2,3}])` returns $\{\}$. Player one has chosen 1, and player two has no legal move. Player one wins the game.

The procedure for computing the list of winning moves from the current position is `WMa(n,P,A,B)`. This procedure is not used in the computation of Sprague-Grundy values, but it provides a way to determine the correct sequence of move for a player who is in a winning position. Some examples from the game traced in the previous example:

- `WMa(10,[10,{}],{1,2},{1,2})` returns $\{\}$. Player one does not have an initial selection from which they can win the game without an error by player two.

- `WMa(10,[3,{3}],{1,2},{1,2})` returns $\{[1,\{1,3\}],[4,\{3,4\}],[5,\{3,5\}]\}$. Player two would have been in a winning position had they chosen any legal option other than 2.

The procedure for computing the Sprague-Grundy value of a position is `Ga(n,P,A,B)`. If P is in the form $[n,\{\}]$, this procedure computes the **mex** of the positions representing an initial selection. Otherwise, it computes the **mex** of $Gh(n,P_i,A,B)$ for all $P_i \in Ma(n,P,A,B)$. Continuing with the above example:

- `Ga(10,[10,{}],{1,2},{1,2})` returns 0. Player one does not have an initial selection from which they can win the game without an error by player two, so this position is a P-position.

Table 4.2: Output of `Magraph(10,{1,2},{1,2})`

Index	Entry
1	$\{2,3\}$
2	$\{1,3,4\}$
3	$\{1,2,4,5\}$
4	$\{2,3,5,6\}$
5	$\{3,4,6,7\}$
6	$\{4,5,7,8\}$
7	$\{5,6,8,9\}$
8	$\{6,7,9,10\}$
9	$\{7,8,10\}$
10	$\{8,9\}$

- `Ga(10,[3,{3}],{1,2},{1,2})` returns 2. This value is positive, so this position is an N-position. Player two has winning options for which the SG value of the ensuing position is 0 (selecting 1, 4, or 5) and the losing option where the SG value of the ensuing position is 1 (selecting 2).

As in the previous case, the improved implementation is principally designed to produce Sprague-Grundy values. There is no procedure for computing winning moves, although one could use the existing procedures to find moves with positive Sprague-Grundy values to accomplish that task. This implementation does not recompute the differences $c-a$ and sums $c+b$ at every stage. Instead, the procedure `Magraph(n,A,B)` produces and stores an adjacency table for the game. See Table 4.2 for an example.

The procedure `Gagraph1(n,A,B,c,S)` computes the Sprague-Grundy value for a position in the game with n numbers, current selection c, and set of numbers already selected S. It first produces a (possible empty) list of legal moves by subtracting S from the entry in the adjacency table corresponding to c. If there are no legal moves, the procedure returns 0. If there are legal moves, we return the **mex** of the Sprague-Grundy values of those moves from the current position.

The procedure `Gagraph(n,A,B)` returns the Sprague-Grundy value of the initial position for a board with n spaces and player one about to make the initial selection. It first calls `Magraph` to produce the adjacency graph, then finds the **mex** of the Sprague-Grundy values of every initial selection $1 \leq c \leq n$ using `Gagraph1(n,A,B,c,{c})`.

4.4 Original Game

As described previously, Lemoine has determined completely for which values of n Juniper Green is winnable by a player one using optimal strategy. We are interested in whether the Sprague-Grundy values of the initial positions for each n exhibit some pattern. In particular, we hope to discover whether the Sprague-Grundy values are eventually periodic.

4.4.1 Pruning Attempt

It seems at first glance that we should be able to employ Lemoine's pruning method described in Section 4.2.1. If we remove the number 1 from our game along with every prime $p > \frac{n}{2}$, the total number of game positions would be significantly reduced.

Deleting vertex 1 and any vertices which are now isolated from the adjacency graph does not change whether or not the game is winnable by player 1. Unfortunately, it does change the Sprague-Grundy values of the initial positions. That is, any Sprague-Grundy value that is 0 will remain 0, but non-zero Sprague-Grundy values can change.

One small case which illustrates this discrepancy is $n = 8$. The adjacency graph for $n = 8$ is shown in Figure 4.2. This case is small enough that we can compute its Sprague-Grundy value by hand.

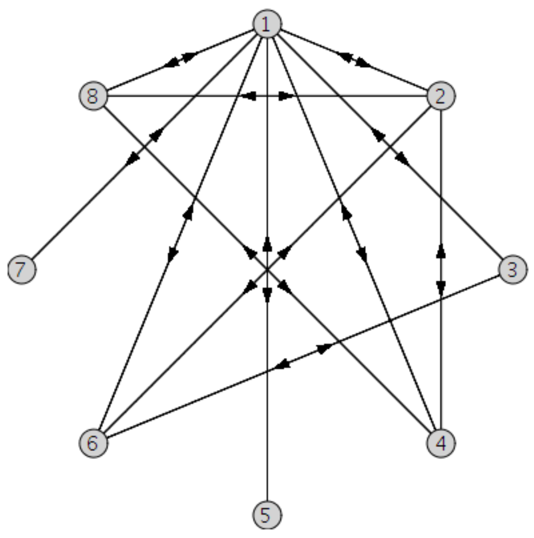

Figure 4.2: Adjacency graph for $n = 8$

If player one selects 2, 4, or 8, then player two has the following options:

- They can select 1, in which case they lose immediately when player one chooses 5 or 7.
- They can select one of the remaining powers of 2. In this case, they lose two moves later when player one selects the other power of 2, forcing player two to take 1.
- In the specific case that one of player one's selections is 2, player two could instead choose 6. Player one can respond to that by choosing 3, again forcing player two to select 1.

Therefore, the Sprague-Grundy values for the position where player one has initially selected 2, 4, or 8 are all 0 (no winning option).

If player one selects 6 initially, there are more possibilities. Player 2 can now choose 2 or 3 (which are winning choices - the Sprague-Grundy value for player one after selections [6,2] or [6,3] is 0). Alternatively, player two can select 1 (a losing

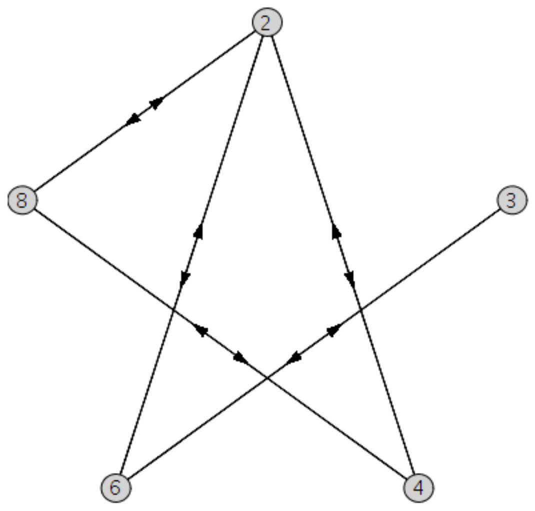

Figure 4.3: Reduced adjacency graph for $n = 8$

option - the SG value after selections [6,1] is 1). So the Sprague-Grundy value for the position where player one has initially selected 6 is $\mathbf{mex}(\{0,1\}) = 2$. This means that the Sprague-Grundy value of the game (before the opening move) is $\mathbf{mex}(\{0,2\}) = 1$.

If we prune vertex 1 from the adjacency graph, we get the reduced form shown in Figure 4.3. Here, if player one selects 2, 4, or 8 initially, we are in essentially the same case as above. The only difference is that instead of being forced to choose 1, player two is left with no legal move one turn earlier.

The discrepancy arises from the case that player one (incorrectly) chooses 6 as the opening move. Now, player two actually has no losing option. They can choose 3 and win immediately, or they can choose 2 and win one turn later. We have effectively removed a possible mistake for player two. The Sprague-Grundy value of the position after an initial selection of 6 is therefore 1.

At this point, we see that instead of having $\mathbf{mex}(\{0,2\}) = 1$ for the value of the game, we have $\mathbf{mex}(\{0,1\}) = 2$. Although the game is still a forced win for player one using optimal strategy, the Sprague-Grundy value has changed. As a result, we can not use this pruning method to explore Sprague-Grundy values.

4.4.2 Results

Using the methodology presented above, we are able to produce the Sprague-Grundy values for $1 \leq n \leq 35$. These values are

$$[0,0,1,0,0,0,0,1,0,0,0,2,2,2,0,1,1,0,0,1,0,0,0,1,0,0,2,0,0,2,2,0,2,2,0].$$

The cases above $n = 30$ take a long time to run, even with the improved speed resulting from storing the adjacency table. The exponential growth in the tree of game positions quickly overwhelms the elimination of the duplicate calculations. We know from Lemoine's results that the last 0 in the sequence of Sprague-Grundy values appears at $n = 118$, so any attempts to discern periodic behavior would need to extend well beyond that case. While further optimization of the current algorithm is possible, it seems unlikely that any direct computation of Sprague-Grundy values which recursively churns through game positions will be able to approach solving cases above $n = 118$.

We may explore other algorithms in the future to try and compute Sprague-Grundy values in some indirect fashion. For now, we abandon the attempt to find periodicity in the values for the original game and turn our attention to variations which alter the manner in which the legal moves are determined.

4.5 Factor Restriction Variation

As in the basic game, we have a board numbered with the integers from 1 to n. In this variation, we have two additional game parameters, positive integers a and b.

Player 1 starts the game by selecting an even number on the board. Players then alternate selecting previously unused numbers which are factors or multiples of the current number c, with the additional restriction that the largest permissible divisor is a and the largest permissible multiplier is b. More precisely, the set of legal moves from a position where c is the current selection is

$$(\{x : x = c/a_1, a_1 \leq a\} \cup \{x : x = c * b_1, b_1 \leq b\}) \setminus \{x : x \text{ has been selected previously}\}$$

4.5.1 Symmetric Cases

In this section, we consider the cases where $A = B$. For the simplest of these cases, $A = B = 2$, we see the following pattern of Sprague-Grundy values:

$$[1, 0, 2, 2, 2, 2, 2, 3, 3, 3, 3, 3, 3, 3, 3, 2, 2, 2, 2, 2, 2, 2, 2, 3, 3, 3, 3, 3, 3, 3].$$

We further note that the Sprague-Grundy value remains 3 for all $n > 30$, having tested cases up to $n = 600$. This case is trivial in the sense that once player one has made the initial selection, there are only two possible ways the game can proceed: player two can either divide by two or multiply by two. After that point, each player has as most one legal option until one end of the board is reached.

For $A = B = 3$, each player is permitted to multiply or divide the current number by 2 or 3. The first 200 Sprague-Grundy values are:

$$[1, 0, 2, 0, 3, 2, 2, 3, 3, 3, 3, 3, 3, 3, 3, 2, 2, 3, 3, 3, 3, 3, 3, 3, 3, 3, 4, 4, 4, 4, 4, 3, 3, 3, 3, 3, 3, 3, 3, 3,$$
$$3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3,$$
$$3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3,$$
$$3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 3,$$
$$3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4]$$

Starting at $n = 18$, we see alternating blocks of 3's and 4's. The blocks of 3's are of lengths 9, 22, 71, and 29. The blocks of 4's are of length 5, 10, 25, and 54. We suspect that these blocks will continue to alternate, but we do not have a conjecture for the block lengths.

For $A = B = 4$, each player is permitted to multiply or divide the current number by 2, 3, or 4. The first 100 Sprague-Grundy values are:

$$[1, 0, 2, 0, 3, 1, 1, 3, 3, 3, 3, 4, 4, 4, 4, 2, 2, 3, 3, 3, 3, 3, 3, 3, 3, 3, 4, 4, 4, 4, 4, 3, 3, 3, 3, 3, 3, 3, 3, 3,$$
$$3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 3,$$
$$3, 3, 3, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4]$$

Starting at $n = 18$, we see alternating blocks of 3's and 4's. The blocks of 3's are of lengths 9, 28, and 4. The blocks of 4's are of length 5, 20, and 28. Once again, we suspect that these blocks will continue to alternate, but we do not have a conjecture for the block lengths.

4.5.2 Asymmetric cases

In this section, we consider the cases where $A \neq B$. For $A = 2, B = 3$, we see the following pattern of Sprague-Grundy values for $1 \leq n \leq 200$:

$$[1, 0, 2, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 4, 4, 4, 4, 4, 3, 3, 3, 3, 3, 3, 3, 3, 3,$$
$$3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3,$$
$$4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3,$$
$$3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 3,$$
$$3, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4]$$

Here, the blocks of 3's and 4's start at $n = 4$. It appears that the Sprague-Grundy value remains 4 for all $n \geq 162$, which we have confirmed for $162 \leq n \leq 500$.

For $A = 3, B = 2$, we see the following for $1 \leq n \leq 80$:

$$[1, 0, 2, 2, 2, 2, 2, 3, 3, 3, 3, 3, 3, 3, 3, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 4, 4, 4, 4, 4, 4, 4, 4, 4,$$
$$4, 4, 4, 4, 4, 4, 4, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3]$$

The block of 3's at the end of that list runs from $n = 48$ to $n = 127$. It is followed by a block of 4's from $n = 128$ to $n = 335$, after which there is another block of 16 3's, and then a block of 4's which extends at least to $n = 500$.

4.6 Additive Variations

As in the basic game, we have a board numbered with the integers from 1 to n. In this variation, we have two additional game parameters, sets of positive integers A

and B. Player 1 starts the game by selecting any number on the board. We do not restrict the initial selection to an even number. Divisibility is relevant in the basic game, but it is not relevant in the additive scenario. Players then alternate selecting previously unused numbers by subtracting an element of A from the current number c or adding an element of B to c. That is, the legal moves from a position where c is currently selected are

$$(\{x : x = c - a, a \in A\} \cup \{x : x = c + b, b \in B\}) \setminus \{x : x \text{ has been selected previously}\}$$

4.6.1 Equal Singleton Sets

In the case where A and B each contain the same single integer, say i, the analysis is relatively simple. In all such games, the first player chooses any number c to start the game. The second player then has at most two options: $c - i$ and $c + i$. Neither player has any options after that point and must continue in the same direction. We can therefore compute the Sprague-Grundy value of a game by considering each starting value in turn and calculating which player wins after each of the two possible responses by player two. It is clear that the Sprague-Grundy value of the position after the initial selection can be at most 2, since there are only two legal moves from that position. Therefore, we know that the Sprague-Grundy value of the game before the initial selection is at most 3. These cases are simple enough that we can compute these values by hand. We will denote the Sprague-Grundy value of the game with n numbers and $A = B = \{m\}$ as $G_m(n)$.

In the specific case $A = B = \{1\}$, we reason as follows. For $n = 1$, the first player selects 1 and wins. The SG value after the selection is 0, so $G_1(1) = 1$. For $n = 2$, the first player selects a number, the second player selects the other, and player two wins. The SG value after either initial choice is 1, so we have $G_1(2) = 0$. For any odd

$n \geq 3$, the first player can select either an odd number or an even number. An odd selection leaves an even number of numbers remaining on each side, so the first player will win; the SG value after choosing an odd number is 0. An even selection leaves an odd number remaining, so the second player will always win; the SG value after choosing an even number is 1. Therefore, $G_1(n) = 2$ for any initial game with odd $n > 3$. For any even $n \geq 4$, any selection by the first player will leave an odd number of moves remaining on one side and an even number on the other. The second player can always win the game by moving in the direction with an odd number of moves, so we must have $G_1(n) = 0$ for even $n \geq 4$. We therefore expect that the sequence of Sprague-Grundy values for this game is $1, 0, 2, 0, 2, 0, 2, 0, \ldots$. This sequence is confirmed by our Maple program.

In the more general case $A = B = \{m\}$, while each game is still completely determined by the first two selections, the analysis is slightly more involved. For all $n \leq m$, the first player selects any number and wins immediately, so $G_m(n) = 1$ for $1 \leq n \leq m$. When $m < n < 2m$, the first player can either select a number close enough to $\frac{n}{2}$ which will result in an immediate win or a number farther away from $\frac{n}{2}$ which results in the second player having a legal (and winning) move, so $G_m(n) = 2$ in these cases. For the specific case $n = 2m$, the first player's initial selection will always leave exactly one legal move for the second player, so $G_m(2m) = 0$. For any $2m < n < 3m$, either the first player chooses a number near $\frac{n}{2}$ which results in the second player having a legal (and winning) move or a number towards the ends of the range which allows the game to consist of exactly two more moves and a first player win. Again, we have $G_m(n) = 2$ in these cases.

For larger values of n, there are often initial moves for the first player which allow the second player to have both winning and losing options. If $n = 2km$ for some integer k, every initial move leaves the second player with options resulting in an even (possibly zero) number of moves to complete the game in one direction, and an odd number in the other. Since the second player always has a winning option, $G_m(2km) = 0$.

If $n = (2k+1)m$ for some integer k, then there will be blocks of length m at each end of the board such that if player one starts with a selection in that block, the game will consist of zero moves in one direction or $2k$ moves in the other direction. Both of these are wins for player one. As the initial selections move toward the middle of the board, there will be alternating blocks of length m for which both directions have an odd number of remaining moves (player two has only winning options) and blocks of length m for which both directions have an even number of moves (player two has only losing options). Therefore, $G_m((2k+1)m) = 2$.

If $n \mod m \neq 0$, there will be positions representing all four of the possible scenarios: even in both directions, odd in both directions, even to the left and odd to the right, even to the right and odd to the left. These latter two cases give player two both winning and losing options, so the Sprague-Grundy value after the initial selection is 2 for these cases. The first two cases have Sprague-Grundy values of 0 and 1 respectively. Therefore, $G_m(n) = \mathbf{mex}(\{0,1,2\}) = 3$ when $n \mod m \neq 0$.

We can summarize these findings as follows: the sequence of Sprague-Grundy values for $A = B = \{m\}$ will be

$$1^m \, 2^{m-1} \, 0 \, 2^{m-1} \left(2 \, 3^{m-1} \, 0 \, 3^{m-1}\right)^*$$

where the exponents indicate the number of times a particular Sprague-Grundy value repeats. The $*$ after the final parenthesis indicates that the values inside the parentheses repeat for all subsequent values of n.

4.6.2 Unequal Singleton Sets

When $A = \{a\}$ and $B = \{b\}$ are distinct singleton sets, the gameplay is somewhat more complicated than the $A = B$ case. One obvious difference is that for any initial selection which is not near 1 or n, player one will have two options at their second turn. Even with this additional flexibility of movement, however, there will come a point in any game starting near the middle of the range where the game reduces to some smaller case. If at any point, there are $\max(a, b)$ consecutive numbers which have already been selected, the remainder of the game will necessarily be played entirely on one side of that group. Effectively, this means that the remaining moves are isomorphic to a game with a smaller value of n where the current selection is near one of the endpoints of the range.

One other observation is that the sequences of Sprague-Grundy values for $A = \{a\}, B = \{b\}$ and $A = \{b\}, B = \{a\}$ must be identical. This occurs because an initial selection of c by player one in the former case is isomorphic to an initial selection of $n - c + 1$ in the latter case. Therefore, without loss of generality, we will only report results in this section for $A = \{a\}, B = \{b\}$ with $a < b$.

Table 4.3: Sprague-Grundy Values with $A = \{1\}, B = \{b\}, b > 1$

b	Initial Segment	Cycle
2	$[1, 2, 1, 2, 3, 3, 2, 3, 2]$	$[3]$
3	$[1, 2, 2, 0, 2]$	$[3, 3, 0, 3]$
4	$[1, 2, 2, 2, 1, 2, 3, 3, 3, 3, 2, 3, 3, 3, 2]$	$[3]$
5	$[1, 2, 2, 2, 2, 0, 2]$	$[3, 3, 3, 3, 0, 3]$
6	$[1, 2, 2, 2, 2, 2, 1, 2, 3, 3, 3, 3, 3, 3, 2, 3, 3, 3, 3, 3, 2]$	$[3]$
7	$[1, 2, 2, 2, 2, 2, 2, 0, 2]$	$[3, 3, 3, 3, 3, 3, 0, 3]$

Table 4.3 presents the results for $A = \{1\}$ and $B = \{k\}$ for $2 \leq k \leq 7$. These results suggest two conjectures. For k even, the sequence of Sprague-Grundy values is eventually constant with value 3. For k odd, the sequence of Sprague-Grundy values is periodic, with the cycle consisting of k 3's and a 0.

Table 4.4 presents the results for $A = \{2\}$ and $B = \{k\}$ for $3 \leq k \leq 6$.

Table 4.4: Sprague-Grundy Values with $A = \{2\}, B = \{b\}, b > 2$

b	Initial Segment	Cycle
3	$[1, 1, 2, 2, 1, 2, 2, 3, 3, 3, 3, 2, 3, 3, 2]$	$[3]$
4	$[1, 1, 2, 2, 2, 1, 2, 2, 3, 3, 3, 3, 3, 2, 3, 3, 3, 2]$	$[3]$
5	$[1, 1, 2, 2, 2, 2, 1, 2, 2, 3, 3, 3, 3, 3, 3, 2, 3, 3, 3, 3, 2]$	$[3]$
6	$[1, 1, 2, 2, 2, 2, 2, 0, 2, 2]$	$[3, 3, 3, 3, 3, 0, 3, 3]$

4.6.3 Sets of the Form {1,k}

If we set $A = B = \{1, k\}$ for $k > 1$, we see some initial segment of Sprague-Grundy values followed by eventual periodic behavior. These results are summarized in Table 4.5.

Table 4.5: Sprague-Grundy Values with $A = B = \{1, k\}$

A=B	Initial Segment	Cycle
$\{1, 2\}$	$[1]$	$[0, 1]$
$\{1, 3\}$	$[1]$	$[0, 2]$
$\{1, 4\}$	$[1, 0, 2]$	$[0, 1]$
$\{1, 5\}$	$[1]$	$[0, 2]$
$\{1, 6\}$	$[1, 0, 2, 0, 2]$	$[0, 1]$
$\{1, 7\}$	$[1]$	$[0, 2]$

All of these sequences start with a 1 for the trivial case $n = 1$ in which player one selects 1 and the game ends immediately. For even k, the initial segment consists of that 1 followed by $\frac{k}{2} - 1$ repetitions of $[0, 2]$. After the initial segment, the Sprague-

Grundy values are periodic with cycle $[0, 1]$. For odd k, the initial segment consists only of the 1 from the trivial case, and the cycle $[0, 1]$ begins at $n = 2$.

4.6.4 Sets of the Form {2,k}

If we set $A = B = \{2, k\}$ for $k > 2$, we see some initial segment of Sprague-Grundy values followed by eventual periodic behavior as we do in the previous section. These results are summarized in Table 4.6.

Table 4.6: Sprague-Grundy Values with $A = B = \{2, k\}$

A=B	Initial Segment	Cycle
$\{2, 3\}$	$[1, 1, 2, 0, 1, 0, 3]$	$[0, 1]$
$\{2, 4\}$	$[1, 1, 2, 0, 2]$	$[1, 3, 0, 3]$
$\{2, 5\}$	$[1, 1, 2, 0, 2, 0, 1, 0, 3]$	$[0, 1]$
$\{2, 6\}$	$[1, 1]$	$[2, 0, 2, 2]$
$\{2, 7\}$	$[1, 1, 2, 0, 2, 2, 3, 0, 1, 0, 3, 0, 1, 0, 3]$	$[0, 1]$

4.7 Future Work

The first project that presents itself for consideration is extending several of the cases for the additive variation. As part of this work, we would like to verify some of our conjectures for larger values of n and for a wider variety of sets A and B. In particular, the current version of the Maple code is not efficient enough to generate meaningful amounts of data if A and B are larger than two elements each.

It would also be desirable to find a more efficient way to compute Sprague-Grundy values for the original game in which all factors and multiples are legal moves. It is clear that recursive searching of position graphs will not extend even to $n = 118$ where any search for periodic behavior can begin.

For both of these projects, an auxiliary goal is to generate enough terms of sequences to make them viable candidates for submission to the Online Encyclopedia of Integer Sequences.

Chapter 5

Conclusions

There are several common themes shared by the projects presented in the previous three chapters. In each case, we were interested in counting or using a family of objects for which the number of objects grew exponentially with some parameter. Brute force enumeration was of limited utility, able to handle only a few more cases than hand computation. By using pruning techniques, improving code efficiency, and considering the symmetries of the underlying objects, we were able to mitigate the exponential growth to some extent. Eventually, even these improvements were overwhelmed by the exponential growth rate, limiting the number of cases which could be explored.

In Chapter 2, we were able to discard most of the syntactically valid circuits using a combination of techniques. Circuits and subcircuits which were symmetric about their topmost gate provided a significant reduction in the number of circuits which needed to be tested. We were also able to remove any circuit containing two or more gates which could be collapsed. These improvements allowed us to find a (nearly) complete catalog of minimal circuits for Boolean functions of few variables.

In Chapter 3, most of the improvement on Thanatipanonda's results came from a change of perspective. We were able to shift all of the difficult parts of the computations into our kernel. Instead of enumerating all of the different ways in which subcubes can intersect, we were able to compress the different types of intersections into small enough regions of a matrix that we could fully enumerate those submatrices. We were able to use combinatorial analysis to recover the number of symmetric cases for each intersection configuration and complete the computation symbolically.

In Chapter 4, we were admittedly unsuccessful in finding applicable pruning techniques. The main improvement from the earliest version of our code was the storage of the adjacency graph which reduced the number of mathematical operations significantly at a minimal cost in memory. In addition to our analysis of the original Juniper Green game, we created several new variations which may be of interest for future research projects.

Appendix A

Catalog of Circuits

In the catalogs which follow, each entry is the representative function of a Big Equivalence Class. The first line contains the number of the representative function (as in OEIS sequence A227723 [[OEISd]]) followed by the set of true points for the function. The next line is the encoding of a minimal straight-line program which computes the function.

The catalogs for general functions of up to three variables are included here. The catalogs for general functions of four variables and monotone functions of up to five variables are available on the author's website.

A.1 Functions of 1 variable

```
0: {}
[[1], [FALSE]]


1: {[1]}
[[1], [1]]


3: {[-1], [1]}
```

```
[[1], [TRUE]]
```

A.2 Functions of 2 variables

```
0: {}
[[1], [2], [FALSE]]


1: {[1, 1]}
[[1], [2], [1, 1, 2]]


3: {[1, -1], [1, 1]}
[[1], [2], [1]]


6: {[-1, 1], [1, -1]}
[[1], [2], [4, 1, 2]]


7: {[-1, 1], [1, -1], [1, 1]}
[[1], [2], [5, 1, 2]]


15: {[-1, -1], [-1, 1], [1, -1], [1, 1]}
[[1], [2], [TRUE]]
```

A.3 Functions of 3 variables

```
0: {}
[[1], [2], [3], [FALSE]]


1: {[1, 1, 1]}
```

```
[[1], [2], [3], [1, 1, 2], [1, 3, 4]]


3: {[1, 1, -1], [1, 1, 1]}

[[1], [2], [3], [1, 1, 2]]


6: {[1, -1, 1], [1, 1, -1]}

[[1], [2], [3], [4, 2, 3], [1, 1, 4]]


7: {[1, -1, 1], [1, 1, -1], [1, 1, 1]}

[[1], [2], [3], [5, 2, 3], [1, 1, 4]]


15: {[1, -1, -1], [1, -1, 1], [1, 1, -1], [1, 1, 1]}

[[1], [2], [3], [1]]


22: {[-1, 1, 1], [1, -1, 1], [1, 1, -1]}

[[1], [2], [3], [5, 1, 2], [3, 3, 4], [4, 1, 5], [4, 2, 6]]


23: {[-1, 1, 1], [1, -1, 1], [1, 1, -1], [1, 1, 1]}

[[1], [2], [3], [4, 1, 2], [4, 1, 3], [1, 4, 5], [4, 1, 6]]


24: {[-1, 1, 1], [1, -1, -1]}

[[1], [2], [3], [4, 1, 2], [4, 1, 3], [1, 4, 5]]


25: {[-1, 1, 1], [1, -1, -1], [1, 1, 1]}

[[1], [2], [3], [5, 1, 2], [3, 3, 4], [4, 2, 5]]


27: {[-1, 1, 1], [1, -1, -1], [1, 1, -1], [1, 1, 1]}
```

```
[[1], [2], [3], [4, 1, 2], [1, 3, 4], [4, 1, 5]]


30: {[-1, 1, 1], [1, -1, -1], [1, -1, 1], [1, 1, -1]}
[[1], [2], [3], [1, 2, 3], [4, 1, 4]]


31: {[-1, 1, 1], [1, -1, -1], [1, -1, 1], [1, 1, -1], [1, 1, 1]}
[[1], [2], [3], [1, 2, 3], [5, 1, 4]]


60: {[-1, 1, -1], [-1, 1, 1], [1, -1, -1], [1, -1, 1]}
[[1], [2], [3], [4, 1, 2]]


61: {[-1, 1, -1], [-1, 1, 1], [1, -1, -1], [1, -1, 1], [1, 1, 1]}
[[1], [2], [3], [10, 1, 3], [1, 2, 4], [4, 1, 5]]


63: {[-1, 1, -1], [-1, 1, 1], [1, -1, -1], [1, -1, 1], [1, 1, -1],
        [1, 1, 1]}
[[1], [2], [3], [5, 1, 2]]


105: {[-1, -1, 1], [-1, 1, -1], [1, -1, -1], [1, 1, 1]}
[[1], [2], [3], [4, 1, 2], [4, 3, 4]]


107: {[-1, -1, 1], [-1, 1, -1], [1, -1, -1], [1, 1, -1], [1, 1, 1]}
[[1], [2], [3], [1, 1, 2], [5, 3, 4], [4, 1, 5], [4, 2, 6]]


111: {[-1, -1, 1], [-1, 1, -1], [1, -1, -1], [1, -1, 1], [1, 1, -1],
         [1, 1, 1]}
[[1], [2], [3], [4, 2, 3], [5, 1, 4]]
```

```
126: {[-1, -1, 1], [-1, 1, -1], [-1, 1, 1], [1, -1, -1], [1, -1, 1],
          [1, 1, -1]}
[[1], [2], [3], [4, 1, 2], [4, 1, 3], [5, 4, 5]]

127: {[-1, -1, 1], [-1, 1, -1], [-1, 1, 1], [1, -1, -1], [1, -1, 1],
          [1, 1, -1], [1, 1, 1]}
[[1], [2], [3], [5, 1, 2], [5, 3, 4]]

255: {[-1, -1, -1], [-1, -1, 1], [-1, 1, -1], [-1, 1, 1], [1, -1, -1],
         [1, -1, 1], [1, 1, -1], [1, 1, 1]}
[[1], [2], [3], [TRUE]]
```

Appendix B

Maple Code for Boolean Functions

Additional material is available on the author's website. This web page contains all of the Maple code used for the implementation of this project. It also includes an early version of Chapter 2 as an unpublished standalone paper, the full circuit catalogs, and other related information. Some highlights from the code are included in this appendix.

The code is organized into three files:

BoolFns.txt	Tools for exploration of Boolean functions and DNFs (includes a variant of the Quine-McCluskey method)
CanonicalBF.txt	Routines for finding the canonical representatives for each equivalence class of Boolean functions of a particular number of variables
SLprog.txt	Routines for the generation and evaluation of straight-line programs (a model of Boolean circuits)

B.1 General Boolean Functions

This is the code from `SLprog.txt` which generates the straight-line programs. This procedure implements the logic described in section 2.2.1 to generate the subset of straight-line programs to be checked against the partially complete catalog at each stage.

```
AllSLPng:=proc(n,lo,hi) local g,i,j,vars,rvs,rff,slp,slp1,slp2,slpn,
SLP,SLPn,S1,S2,prefix:
option remember:
g:=hi-lo+1:
print("lo",lo,"hi",hi):
#vars:={seq(k,k=1..n)}: #Changed to allow for reuse of prior gates
vars:={seq(k,k=1..hi-1)}:
if g<1 then print('error'): RETURN(FAIL): fi:
prefix:=[seq([i],i in 1..n),seq([FALSE],j in 1..lo-n-1)]:
SLPn:={}:
SLP:={}:
if g=1 then
  for rff from 1 to 10 do
#    for rvs in choose(n,2) do    #Changed to allow for reuse of prior gates
    for rvs in choose(hi-1,2) do
      slp:=[[rff,op(rvs)]]:
      slpn:=BoolToInt(n,SLPntoBool([op(prefix),op(slp)])):
      if not (slpn in SLPn) then
        SLPn:=SLPn union {slpn}:
        SLP:=SLP union {slp}:
      fi:
    od:
  od:
  RETURN(SLP):
```

```
fi:
for i from 0 to floor((g-1)/2) do
  if i=0 then
   S1:=AllSLPng(n,lo,hi-1):
    for rff from 1 to 10 do
      for slp1 in S1 do
        for j in vars minus {op(2..3,slp1[-1])} do
          slp:=[op(slp1),[rff,j,hi-1]]:
          slpn:=BoolToInt(n,SLPntoBool([op(prefix),op(slp)])):
          if not (slpn in SLPn) then
            SLPn:=SLPn union {slpn}:
            SLP:=SLP union {slp}:
          fi:
        od:
      od:
    od:
  else
   S1:=AllSLPng(n,lo,lo+i-1):
   S2:=AllSLPng(n,lo+i,hi-1):
   for rff from 1 to 10 do
     for slp1 in S1 do
       for slp2 in S2 do
        slp:=[op(slp1),op(slp2),[rff,lo+i-1,hi-1]]:
        slpn:=BoolToInt(n,SLPntoBool([op(prefix),op(slp)])):
        if not (slpn in SLPn) then
          SLPn:=SLPn union {slpn}:
          SLP:=SLP union {slp}:
```

```
        fi:
       od:
     od:
   od:
  fi:
od:
SLP:
end:
```

The following procedures from `CanonicalBF.txt` initialize and then populate the catalog file. For each number of variables n, we run `GenBFCatFileAll(n)` first. We then run `CatFileFindSLPAll(n,g)` successively for $g = 0, 1, 2, \ldots$ until every function has been matched to a straight-line program which computes it.

```
GenBFCatFileAll:=proc(n) local fn,F,f,T:
fn:=cat("BFCatFileA",n,"var.txt"):
F:=FindCanonBFNP(n):
T := table():
for f in F do
    T[BoolToInt(n, f)] := f;
od:
FileTools[Text][WriteLine](fn, cat("Functions: ",convert(nops(F),string),
  " Found: 0")):
for f in indices(T,indexorder) do
  FileTools[Text][WriteLine](fn, cat(convert(op(f),string),": ",
    convert(T[op(f)],string))):
  FileTools[Text][WriteLine](fn, "NF"):
```

```
  FileTools[Text][WriteLine](fn, ""):
od:
FileTools[Text][Close](fn):
fn:
end:



CatFileFindSLPAll:=proc(n,g) local numf,numfd,l1,l2,l3,fn,fnold,A,T,
  F,f,SLP,slp,slpf,i:
fn:=cat("BFCatFileA",n,"var.txt"):
fnold:=cat("BFCatFileA",n,"var-OLD.txt"):
FileTools[Text][Open](fn):
l1:=FileTools[Text][ReadLine](fn):
l2:=sscanf(l1,"%s %d %s %d"):
numf:=l2[2]:
numfd:=l2[4]:
F:={}:
T:=table():
for i from 1 to numf do
  l1:=FileTools[Text][ReadLine](fn):
  if l1=NULL then break: fi:
  l2:=FileTools[Text][ReadLine](fn):
  l3:=FileTools[Text][ReadLine](fn):
  l3:=sscanf(l1,"%d: %s"):
  f:=l3[1]:
  T[f]:=parse(l2):
  if l2="NF" then
```

```
    F:=F union {f}:
  fi:
od:
if nops(F)=0 then RETURN(F): fi:
FileTools[Text][Close](fn):
SLP:=AllSLPn(n,g):
print("Have SLP's - entering loop"):
print(nops(SLP)," - number of SLPs"):
for i from 1 to nops(SLP) while nops(F)>0 do
  slp:=SLP[i]:
  slpf:=BoolToInt(n,SLPntoBool(slp)):
  if T[slpf]=NF then
    F:=F minus {slpf}:
    T[slpf]:=slp:
    numfd:=numfd+1:
  fi:
  if i mod 1000=0 then print(i): fi:
od:
if FileTools[Exists](fnold) then FileTools[Remove](fnold): fi:
FileTools[Rename](fn,fnold):
FileTools[Text][WriteLine](fn, cat("Functions: ",convert(numf,string),
  " Found: ",convert(numfd,string))):
for f in indices(T,indexorder) do
    FileTools[Text][WriteLine](fn, cat(convert(op(f),string),": ",
      convert(IntToBool(n,op(f)),string))):
    FileTools[Text][WriteLine](fn, convert(T[op(f)],string)):
    FileTools[Text][WriteLine](fn, ""):
```

```
od:
FileTools[Text][Close](fn):
F:
end:
```

B.2 Monotone Functions

The following procedures find the set of monotone functions on n variables. This logic checks many fewer functions than the brute-force iteration used for general functions. Without this modification, it would not be possible to generate the list of BEC representatives for functions of five variables.

```
IsMonotone:=proc(n,f) local i,j,v,s,nv:
if nops(f)=0 then RETURN(true): fi:
for v in f do
  s:={}:
  for i from 1 to n do
    if v[i]=-1 then s:=s union {i}: fi:
  od:
  for i in powerset(s) do:
    nv:=v:
    for j in i do nv[j]:=1: od:
    if not (nv in f) then RETURN(false): fi:
  od:
od:
RETURN(true):
end:
```

```
FindMonoBFP:=proc(n) local PT,i,j,v,nv,nv2,f,F,ff,W:
F:={{}}: #Adds empty function because it is only monotone function
#                                             not containing [1$n]
W:={{[1$n]}}: # Adds the AND of all variables as the seed function
#                                                in working set W
while nops(W)>0 do
  ff:=W[1]:  #pull function from W
  if not (ff in F) then
    F:=F union {ff}:
    for v in ff do
      for i from 1 to n do
        if v[i]=1 then
          nv:=[op(1..i-1,v),-1,op(i+1..n,v)]:
          f:=ff union {nv}:
          if not (f in F) and IsMonotone(n,f) then W:=W union {f}: fi:
        fi:
      od:
    od:
  fi:
  W:=W minus {ff}:
od:
ff:={}:
for i in F do
  if IsBECrep(n,i) then
    ff:=ff union {i}:
  fi:
od:
```

```
ff:

end:
```

Appendix C

Maple Code for Subcubes of Boolean Functions

Additional material is available on the author's website. This web page contains all of the Maple code used for the implementation of this project and selected outputs.

Appendix D

Maple Code for Juniper Green

Additional material is available on the author's website. This web page contains all of the Maple code used for the implementation of this project and selected outputs.

The following code finds the Sprague-Grundy values for initial positions of the original version of Juniper Green. The code for the variations discussed in Chapter 4 is similar. The complete Maple package can be found here.

This routine builds the adjacency graph.

```
Mhgraph:=proc(n) local G,i,j,v:
option remember:
G:=table():
for v from 1 to n do
G[v]:={seq(i,i in (divisors(v) union
    {seq(v*i,i=1..trunc(n/v))}) minus {v})}:
od:
op(G):
end:
```

The following pair of procedures finds Sprague-Grundy values. The procedure call to find the Sprague-Grundy value for $n = 100$ in the original game is `Ghgraph(100)`.

```
Ghgraph1:=proc(n,c,S) local N,i:
option remember:
N:=Mhgraph(n)[c] minus S:
if nops(N)=0 then
 0:
else
 mex({seq(Ghgraph1(n,i,S union {i}),i in N)}):
fi:
end:


Ghgraph:=proc(n) local G,v,sv,i,j:
option remember:
sv:={seq(2*j,j in 1..trunc(n/2))}:
mex({seq(Ghgraph1(n,i,{i}),i in sv)}):
end:
```

The above procedures rely on the `mex` function defined by:

```
mex:=proc(A) local i:
if A={} then
 RETURN(0):
fi:
for i from 0 while member(i,A) do od:
i:
end:
```

Bibliography

[BCG01] Elwyn R. Berlekamp, John H. Conway, and Richard K. Guy, *Winning Ways for Your Mathematical Plays*, 2nd ed., Vol. 1, CRC Press, Taylor & Francis Group, 2001.

[Blu84] Norbert Blum, *A Boolean function requiring* $3n$ *network size*, Theoret. Comput. Sci. **28** (1984), no. 3, 337–345.

[BM11] Carl B. Boyer and Uta C. Merzbach, *A History of Mathematics*, Wiley, 2011.

[CH11] Yves Crama and Peter L. Hammer, *Boolean Functions*, Cambridge University Press, 2011.

[GHKK18] Alexander Golovnev, Edward A. Hirsch, Alexander Knop, and Alexander S. Kulikov, *On the limits of gate elimination*, Journal of Computer and System Sciences **96** (2018), 107-119.

[Hal85] Paul R. Halmos, *I Want to Be a Mathematician, an Automathography*, Springer-Verlag, 1985.

[Har65] Michael A. Harrison, *Introduction to switching and automata theory*, McGraw-Hill, 1965.

[JSZ23] Svante Janson, Blair Seidler, and Doron Zeilberger, *On the Statistics of the Number of Fixed-Dimensional Subcubes in a Random Subset of the* n*-Dimensional Discrete Unit Cube*, arXiv:2302.09047 (2023).

[Kon93] Urszula Konieczna, *Asymptotic normality of subcubes in random subgraphs of the* n*-cube.*, Discrete Mathematics **121** (1993), 117-122.

[Lem22] Julien Lemoine, *Résolution du Jeu de Juniper Green*, arXiv:2202.09864 (2022).

[OEISa] *OEIS Foundation Inc. (2021), The On-Line Encyclopedia of Integer Sequences*, `https://oeis.org/A000616`.

[OEISb] *OEIS Foundation Inc. (2021), The On-Line Encyclopedia of Integer Sequences*, `https://oeis.org/A000231`.

[OEISc] *OEIS Foundation Inc. (2021), The On-Line Encyclopedia of Integer Sequences*, `https://oeis.org/A227722`.

[OEISd] *OEIS Foundation Inc. (2021), The On-Line Encyclopedia of Integer Sequences*, `https://oeis.org/A227723`.

[OEISe] *OEIS Foundation Inc. (2021), The On-Line Encyclopedia of Integer Sequences*, `https://oeis.org/A349743`.

[Pau77] Wolfgang J. Paul, *A* 2.5*n-lower bound on the combinational complexity of Boolean functions*, SIAM J. Comput. **6** (1977), no. 3, 427–443. MR451856

[Pie20] Tilman Piesk, *Equivalence Classes of Boolean Functions*, 2020. `https://en.wikiversity.org/wiki/Equivalence_classes_of_Boolean_functions`.

[Qui52] W. V. Quine, *The Problem of Simplifying Truth Functions*, The American Mathematical Monthly **59** (1952), no. 8, 521–531.

[Raz85] A. A. Razborov, *Lower bounds on the monotone complexity of some Boolean functions*, Doklady Akademii Nauk SSSR **281** (1985), 798-801.

[SB99] L. Lynn Stallings and Patricia L. Bullock, *Juniper Green*, Mathematics Teaching in the Middle School **4** (1999), no. 7, 438-440.

[Ste97] Ian Stewart, *Mathematical Recreations: Juniper Green*, Scientific American **276** (1997), 118-120.

[Saa06] Markku-Juhani Olavi Saarinen, *Chosen-IV Statistical Attacks on eStream Ciphers*, SECRYPT 2006, Proceedings of the International Conference on Security and Cryptography, Setúbal, Portugal, August 7-10, 2006, pp. 260–266.

[Tha20] Thotsaporn Aek Thanatipanonda, *Reviews of Symbolic Moment Calculus*, arXiv:2003.11749 (2020).

[Tar88] É. Tardos, *The gap between monotone and non-monotone circuit complexity is exponential*, Combinatorica (Budapest 1981) **8** (1988), no. 1, 141-142.

[Zei04] Doron Zeilberger, *Symbolic Moment Calculus I: Foundations and Permutation Pattern Statistics*, Annal of Combinatorics **8** (2004), 369-378.