5-2011

Biologically Relevant Classes of Boolean Functions

Lori Layne
Clemson University, llayne@g.clemson.edu

Biologically Relevant Classes of Boolean Functions

A Dissertation
Presented to
the Graduate School of
Clemson University

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy
Mathematical Sciences

---

by
Lori Layne
May 2011

Accepted by:
Dr. Elena Dimitrova, Committee Chair
Dr. Brian Dean
Dr. Shuhong Gao
Dr. Matthew Macauley

Abstract

A large influx of experimental data has prompted the development of innovative computational techniques for modeling and reverse engineering biological networks. While finite dynamical systems, in particular Boolean networks, have gained attention as relevant models of network dynamics, not all Boolean functions reflect the behaviors of real biological systems. In this work, we focus on two classes of Boolean functions and study their applicability as biologically relevant network models: the nested and partially nested canalyzing functions.

We begin by analyzing the *nested canalyzing functions* (NCFs), which have been proposed as gene regulatory network models due to their stability properties. We introduce two biologically motivated measures of network stability, the average height and average cycle length on a state space graph and show that, on average, networks comprised of NCFs are more stable than general Boolean networks.

Next, we introduce the *partially nested canalyzing functions* (PNCFs), a generalization of the NCFs, and the *nested canalyzing depth*, which measures the extent to which it retains a nested canalyzing structure. We characterize the structure of functions with a given depth and compute the expected activities and sensitivities of the variables. This analysis quantifies how canalyzation leads to higher stability in Boolean networks. We find that functions become decreasingly sensitive to input perturbations as the canalyzing depth increases, but exhibit rapidly diminishing returns in stability. Additionally, we show that as depth increases, the dynamics of networks using these functions quickly approach the critical regime, suggesting that real networks exhibit some degree of canalyzing depth, and

that NCFs are not significantly better than PNCFs of sufficient depth for many applications to biological networks.

Finally, we propose a method for the reverse engineering of networks of PNCFs using techniques from computational algebra. Given discretized time series data, this method finds a network model using PNCFs. Our ability to use these functions in reverse engineering applications further establishes their relevance as biological network models.

Dedication

To Mom and Dad.

Acknowledgments

Foremost, I would like to thank my adviser, Elena Dimitrova, with whom I have had the pleasure of working for the last five years. Thank you for your guidance, wisdom, and patience, and for setting a great example for me as a research mathematician.

I would like to thank my committee members, Shuhong Gao and Brian Dean, for your helpful comments during the research process and for all you have taught me in and out of the classroom. Thank you to Matthew Macauley for all of your insight and many contributions, especially with the depth and stability work.

Thank you to my parents, without whose tireless love and encouragement this work would not be possible. Thank you Andrew for reminding me to stay balanced. Thank you Frank for all of your computer help over the years and for your friendship. Thank you Mooch and Emily for keeping me grounded during the prelims. Finally, I would like to thank all my friends and family for your confidence and support throughout this entire process.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Discrete Network Models

An accumulation of biological data has prompted the development of new mathematical techniques to process and organize such data. A central problem in systems biology is the modeling and reverse engineering of gene regulatory networks to discover how genes interact via their RNA and protein products to regulate each other, and to explore the dynamics of such networks. Davidson et al. define gene regulatory networks (GRNs) as "collections of genes and their products, together with the interactions between them that collectively carry out cellular functions" [13]. GRNs give insight into causality relationships in the genome [13]. Studying regulatory and transcription networks can lead to a greater understanding of human health and can ultimately help fight disease [41]. While ordinary differential equations have traditionally been used to model dynamical systems, time-discrete finite dynamical systems have gained attention as prominent biochemical network models. Several types of FDS models have been studied in this context, for instance, Petri nets [21], Logical models [53], polynomial models [29, 40], and Boolean networks [2, 42]. Discrete models have been used for a variety of applications in addition to systems biology, some of which include chaos [43], traffic simulations [47], task scheduling on parallel computing systems [46], immunology [7], and control theory [5].

Discrete models tend to be simpler and can be more intuitive than continuous models. Typically, there are no initial conditions or parameters to estimate, which is quite an advantage over their continuous counterparts. Discrete models consider the effects of individual components within the network, not just measuring the network as a whole, so it is possible to observe how altering or perturbing a subset of the components can affect system dynamics. Finally, as we shall see, some discrete models have convenient representations as algebraic structures, allowing us to employ tools and algorithms from algebraic geometry and computational algebra to construct appropriate network models, thus enabling us to examine dynamical properties of the system.

1.2 Computational Algebra Basics

1.2.1 Ideals and Varieties

Ideals and varieties are essential structures in discrete modeling, especially for their utility in reverse engineering. Here, we include several key definitions and properties, as presented in [9].

Definition 1. *Let* $\mathbb{F}$ *be a field and* $f_1, \ldots, f_s \in \mathbb{F}[x_1, \ldots, x_n]$. *Then the set*

$$\mathbb{V}(f_1, \ldots, f_s) = \{(a_1, \ldots, a_n) \in \mathbb{F}^n : f_i(a_1, \ldots, a_n) = 0, 1 \leq i \leq n\}$$

is the affine variety *defined by* $f_1, \ldots, f_s$.

In other words, an affine variety defined by $f_1, \ldots, f_s$ can be thought of as the common roots of $f_1, \ldots, f_s$.

Example 1. *The affine variety* $\mathbb{V}(\frac{x^2}{4} + y^2 - 1)$ *over* $\mathbb{R}^2$ *is given in Figure 1.1.*

Several important relationships exist between ideals and affine varieties. For instance, if $V \subset \mathbb{F}^n$ is an affine variety, then

$$\mathbb{I}(V) = \{f \in \mathbb{F}[x_1, \ldots, x_n] : f(a_1, \ldots, a_n) = 0, \ \forall (a_1, \ldots, a_n) \in V\}$$

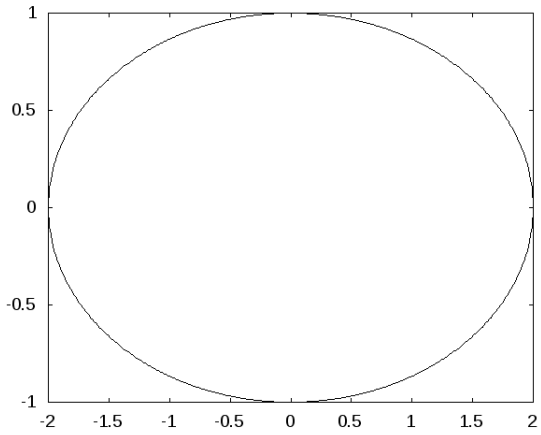

Figure 1.1: $\mathbb{V}\left(\frac{x^2}{4} + y^2 - 1\right)$ in Example 1

is an ideal, called the *ideal of* V. This ideal is simply the set of polynomials that vanish on the points in V. On the other hand, if $I \subset \mathbb{F}[x_1, \ldots, x_n]$ is an ideal, then

$$\mathbb{V}(I) = \{x \in \mathbb{F}^n : f(x) = 0,\ \forall f \in I\}$$

is an affine variety. An important relation between ideals and their associated varieties is the Ideal-Variety Correspondence, a result of Hilbert's well-known Nullstellensatz. For an algebraically closed field $\mathbb{F}$, this correspondence tells us that

1. For $V \in \mathbb{F}^n$, $\mathbb{V}(\mathbb{I}(V)) = V$.

2. If $\mathbb{F}$ is algebraically closed and I is a radical ideal, then $\mathbb{I}(\mathbb{V}(I)) = I$.

In the case of Boolean networks, we will be working with polynomials over $\mathbb{F}_2$, which is not algebraically closed. In Chapter 4, we will see ideals from polynomial rings over $\overline{\mathbb{F}}_2$, the algebraic closure of $\mathbb{F}_2$.

In Chapter 4, we will encounter a class of functions that form a so-called *toric ideal*. A toric ideal may be thought of as the Zariski closure of the image of a monomial map. Toric ideals and their corresponding varieties are well-studied structures in algebraic geometry with computationally desirable properties. Further information on toric varieties and ideals

may be found in [10, 44]. Toric ideals are the binomial prime ideals [19]. Since prime ideals are radical, the Ideal-Variety Correspondence tells us that varieties corresponding to a toric ideals are also toric.

1.2.2 Gröbner Bases

Gröbner basis computation is a critical step in the reverse engineering algorithms that we will encounter in Chapter 4. Here, we present several definitions and properties associated with Gröbner bases which may be found in [9, 58]. A Gröbner basis is dependent upon its so-called monomial ordering. Note that we can represent a monomial $x_1^{\alpha_1} \dots x_n^{\alpha_n}$ in $\mathbb{F}[x_1, \dots, x_n]$ by its exponents as $\alpha = (\alpha_1, \dots, \alpha_n) \in \mathbb{N}^n$. We can now formally define a monomial ordering as follows.

Definition 2. *A* monomial ordering *on* $\mathbb{F}[x_1, \dots, x_n]$ *is a relation* $<$ *on* $\mathbb{N}^n$ *such that*

1. $<$ *is a total ordering on* $\mathbb{N}^n$.
2. *If* $\alpha, \beta, \gamma \in \mathbb{N}^n$ *with* $\alpha < \beta$, *then* $\alpha + \beta < \alpha + \gamma$.
3. $<$ *is a well-ordering on* $\mathbb{N}^n$.

Example 2. *One common monomial ordering is* lexicographic order, *which can be considered an alphabetical ordering. For example, under lexicographic order with* $x > y > z$, *we have* $xyz > xz$.

Definition 3. *Let* $f = \sum_{\alpha \in \mathbb{N}^n} a_\alpha x^\alpha \in \mathbb{F}[x_1, \dots, x_n]$ *be nonzero and* $<$ *a monomial order. Then*

1. *The* multidegree *of* f *is* $multideg(f) = max_<\{\alpha \in \mathbb{N}^n : a_\alpha \neq 0\}$.
2. *The* leading coefficient *of* f *is* $LC(f) = a_{multideg(f)} \in \mathbb{F}$
3. *The* leading monomial *of* f *is* $LM(f) = x^{multideg(f)}$.
4. *The* leading term *of* f *is* $LT(f) = LC(f) \cdot LM(f)$.

Finally, for an ideal $I \subseteq \mathbb{F}[x_1, \ldots, x_n]$, $\mathrm{LT}(I)$ is the set of leading terms of polynomials in I, and $\langle \mathrm{LT}(I) \rangle$ is the ideal generated by $\mathrm{LT}(I)$. We can now formally define a Gröbner basis.

Definition 4. *Let $<$ be a monomial order and $I \subseteq \mathbb{F}[x_1, \ldots, x_n]$ be nonzero. Then a subset $G = \{g_1, \ldots, g_t\}$ is a Gröbner basis for I if $\langle LT(g_1), \ldots, LT(g_t) \rangle = \langle LT(I) \rangle$.*

The well-known Hilbert Basis Theorem tells us that this basis exists and is finitely generated.

Example 3. *Let $I = \langle f_1,\ f_2 \rangle \in \mathbb{Q}[x, y]$, with $f_1 = x$ and $f_2 = x^2 + y$. Using lexicographic order with $x > y$, $y = -x \cdot f_1 + 1 \cdot f_2$, so $y \in \langle LT(I) \rangle$. However, y is divisible by neither x nor x^2, so $y \notin \langle LT(f_1),\ LT(f_2) \rangle$. Therefore I is not a Gröbner basis for this ideal.*

Even with a fixed monomial ordering, polynomial division is not unique, as it depends on the order of the divisors.

Example 4. *Let $F = \{f_1 = y^2 + 1, f_2 = xy + 1\} \in \mathbb{Q}[x, y]$ and $f = 2xy^2 + x - y$. Using lex order with $x > y$ and the division algorithm for multivariate polynomials in [9], if we divide by f_1 and then f_2, we obtain a remainder of $-x - y$; however, if we reverse the order of the divisors, our remainder is $x - 3y$.*

If we are dividing by a Gröbner basis, however, our remainder is unique regardless of the order of the divisors. The *normal form* of a polynomial $f \in \mathbb{F}[x_1, \ldots, x_n]$ with respect to an ideal $I \subset \mathbb{F}[x_1, \ldots, x_n]$ is the remainder when dividing f by G, where G is the Gröbner basis for I. This normal form is unique up to monomial order, and f lies in I if and only if the normal form for f is zero.

Gröbner basis computation is still an active area of research. The first algorithm for doing so is known as Buchberger's Algorithm, first introduced in 1965. While the worst-case computational complexity for computing Gröbner bases is unknown, it is thought to be exponential [45], although several speedups and special cases exist. For instance, an application of the Buchberger-Möller algorithm yields a fairly efficient computation for a

Gröbner basis of an ideal of points, a special case that we will encounter in Chapter 4 [1]. Newer Gröbner basis algorithms and speedups have been developed, as in [20, 22, 24], some of which are used by current computer algebra systems.

In addition to their utility in discrete modeling, Gröbner bases have various applications in computational algebra. For instance, they are used to solve multivariate systems of polynomial equations, to determine whether a polynomial belongs to a given ideal [9], to determine whether or not two sets of polynomials give rise to the same ideal, and for automatic theorem proving in geometry [58].

1.3 Boolean Networks

Definition 5. *A finite dynamical system is a mapping*

$$F = (f_1, f_2, \ldots, f_n) : X^n \to X^n$$

where X is a finite set.

The elements of X are the possible states in which the components (nodes) of the network can lie. The function $f_i(x_1, \ldots, x_n) : X^n \to X$ gives the dynamics of the ith component, $1 \leq i \leq n$, while the variables $x_1, \ldots, x_n$ represent the individual components. For example, in a gene regulatory network model, the components are the individual genes and the elements of X are the discretized gene expression levels. If $X = \mathbb{F}_2$ then the functions f_i are polynomials in $\mathbb{F}_2$ of degree at most one, which can be expressed as Boolean functions. For gene regulatory networks modeled as Boolean networks, $x_i = 0$ implies that gene i is not expressed (OFF), while $x_i = 1$ means that gene i is expressed (ON).

Boolean network models have two key elements, the wiring diagram and the state space graph. The *wiring diagram*, otherwise referred to as the *dependency graph*, specifies which components in the network influence each other. It is visualized as a digraph, where each variable x_i in the system is represented by a node, and an edge from x_j to x_i indicates x_j influences x_i, that is, f_i is a function of x_j.

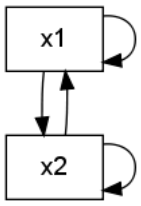

Figure 1.2: Wiring Diagram for Example 5 [38]

Example 5. *The wiring diagram for the network $F = (f_1, f_2)$ is given in Figure 1.2, where*

$$\begin{aligned} f_1 &= 1 + x_1 + x_2 \\ f_2 &= x_1 x_2. \end{aligned}$$

For simplicity, we will consider networks in which every node in the wiring diagram has the same number of inputs; however, in some cases, the in-degrees of the nodes may be selected according to a probability distribution. For example, networks constructed according to the power-law distribution, also known as scale-free networks, have been studied extensively in this context, as in [36, 3].

The *state space graph*, also known as the *phase space graph* is a digraph that conveys the dynamics of a finite dynamical system. The nodes of a state space graph are strings of n bits, with each bit representing the Boolean state of its corresponding component. Note that the state space graph for a network with n components has 2^n nodes. An edge from node $\mathbf{x}$ to node $\mathbf{y}$ indicates that $F(\mathbf{x}) = \mathbf{y}$, when $f_1, f_2, \ldots, f_n$. These functions $f_1, \ldots, f_n$ are called *state transition functions*. We will consider finite dynamical systems with *parallel updating*, in which all of the n state transition functions are evaluated in each time step. When $f_1, \ldots, f_n$ are not updated simultaneously, the system is called an FDS with *sequential updating* [46].

Throughout this work, we will typically let n denote the number of components in the system. In Chapter 3, we will be working networks such that every component has a fixed number of input variables, denoted by k.

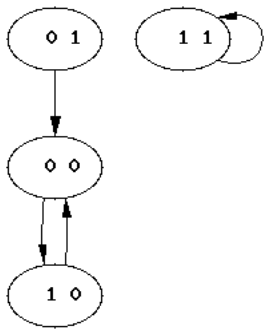

Figure 1.3: State space graph for Example 6 [38]

Example 6. *The state space graph for the network $F = (f_1, f_2)$ in Example 5 is given in Figure 1.3.*

Since the state space is finite, each directed path terminates in a cycle, called a *limit cycle* or an *attractor cycle*. If a limit cycle consists of a single node, it is called a *fixed point*. Nodes that are not part of limit cycles are called *transient states*, and connected components are referred to as *basins of attraction* [39]. Example 6 has two basins of attraction, one a single fixed point, and another with one transient state and a limit cycle of size two. We will consider *deterministic Boolean networks*, for which there is one state transition function governing the dynamics of each component. Consequently, there is a unique path from each node to its terminal limit cycle in the state space graph of a deterministic Boolean network. In the case of *probabilistic Boolean networks*, multiple state transition functions exist for each node. At each time step, the function used in the system update are selected probabilistically from the designated possibilities. In this context, Boolean network can be considered a Markov chain [54].

Of significant interest in determining the biological relevance of a class of Boolean functions is to establish the stability of networks comprised of such functions. While many characterizations of stability exist, we often say that a network is stable if it is insensitive input perturbations, thus determining whether it lies in the frozen, chaotic, or critical phase. Networks in the *frozen* phase are insensitive to small perturbations. The state spaces of

these networks are characterized by small limit cycles and fixed point cycles. Networks in the *chaotic* phase typically have longer limit cycles in their state space graph, and perturbations propagate throughout the network. The *critical* phase is the threshold between the frozen and chaotic phases. It is thought to be the phase in which many biologically networks lie, as they must be stable enough to resist environmental change, yet capable of undergoing essential adaptations [4, 49, 50, 57]. We will explore alternative definitions of stability in Chapter 2.

1.4 Modeling Considerations

While discrete models have advantages over their continuous counterparts in certain situations, a few complications arise when using Boolean networks to model biological systems. For instance, gene regulatory network models built from continuous gene expression data necessitate the development of data discretization methods, such as in [16]. While discretization seemingly leads to a loss of information, reducing the system to two states reflects the observed behavior of real gene networks [33, 2]. Also, using coarser modeling techniques such as Boolean networks reduces the effects of noise in the data [55]. Hartemink justifies the practice of discretization as follows [25].

> Inside cells, biochemical reactions are at the lowest level discrete events in which individual molecules and enzymes are brought together for oxidation, reduction, hydrolysis, catalysis, etc. Given current measurement technology, however, it is impractical to measure whole-genome expression levels at single-molecule resolution. For this reason, large numbers of cells are pooled together and mRNA removed from the population as a whole. Consequently, the various species of mRNA are typically present in sufficient abundance to be represented as continuous concentration values. Nevertheless, reasoning about continuous concentration values can be problematic given the number of degrees of freedom inherent in arbitrary continuous distributions. Because the amount of data available for reasoning about genetic regulatory networks is

comparatively limited, we need to reduce the dimensionality of the modeling.

Another prominent problem in the application of Boolean models is that of selecting state transition functions whose behavior mimics real biological systems. Random Boolean networks were initially introduced by Kauffman [33, 34] as gene network models. In this setup, input variables are randomly selected for each node, (i.e., the wiring diagram is randomly wired), and each component is assigned a random state transition function according to a specified probability distribution. As not all Boolean functions exhibit biological behavior, specific biologically motivated classes of functions have since been introduced, with the hope that restricting the model selection to such functions will result in networks with more appropriate dynamic behaviors. For instance, the chain functions [23], the biologically meaningful functions [52], and the nested canalyzing functions [35] have all been proposed due to their biologically relevant properties.

In this work, we will focus on two particular classes of Boolean functions, the *nested canalyzing functions* (NCFs), and a natural relaxation of NCFs, the *partially nested canalyzing functions* (PNCFs). In Chapter 2, we will discuss the NCFs in depth and explore biologically relevant dynamic properties of the functions themselves, as well as networks constructed using these functions. In Chapter 3, we will define the PNCFs and see how varying the extent to which functions exhibit the nested canalyzing property affects network dynamics. Finally, in Chapter 4, we will introduce a reverse engineering method for gene regulatory networks that restricts the modeling space to PNCFs.

Chapter 2

Biologically Meaningful Properties of Nested Canalyzing Functions

2.1 Introduction

Canalyzing functions were introduced by Kauffman [33], which reflect the behavior of biological systems described by Waddingdon [59]. Specifically, Waddington introduced the term "canalization" to describe the ability of a genotype to yield the same phenotype despite variations in the environment. Mathematically speaking, a function $f(x_1, \ldots, x_n)$ is said to be *canalyzing* if there exists a variable x_i, $1 \leq i \leq n$ and values $a, b \in \{0, 1\}$ such that $f(x_1, \ldots, x_{i-1}, a, x_{i+1}, \ldots, x_n) = b$ regardless of the inputs to the other variables. In [35], the authors introduce the nested canalyzing functions (NCFs), which specify how to proceed if the canalyzing variable x_i does not receive its canalyzing input a. Nested canalyzing functions are defined as follows:

Definition 6. *Let $f(x_1, \ldots, x_n)$ be a Boolean function. For $\sigma \in S_n$, f is a* nested canalyzing function *in the variable order $x_{\sigma(1)}, \ldots, x_{\sigma(n)}$ with canalyzing values $a_1, \ldots, a_n$ and canalyzed*

values $b_1, \ldots, b_n$ if it can be expressed in the form

$$f = \begin{cases} b_1 & x_{\sigma(1)} = a_1 \\ b_2 & x_{\sigma(1)} \neq a_1,\ x_{\sigma(2)} = a_2 \\ b_3 & x_{\sigma(1)} \neq a_1,\ x_{\sigma(2)} \neq a_2,\ x_{\sigma(3)} = a_3 \\ \vdots & \vdots \\ b_n & x_{\sigma(1)} \neq a_1, \ldots,\ x_{\sigma(n-1)} \neq a_{n-1},\ x_{\sigma(n)} = a_n \\ \neg b_n & x_{\sigma(1)} \neq a_1, \ldots,\ x_{\sigma(n)} \neq a_n \end{cases} . \tag{2.1}$$

Example 7. *The function $f(x_1, x_2, x_3) = x_2 \vee (\neg x_1 \wedge x_3)$ is an NCF with $b = [1, 0, 0]$, $a = [1, 1, 0]$, and $\sigma = [2, 1, 3]$:*

$$f(x_1, x_2, x_3) = \begin{cases} 1 & x_2 = 1 \\ 0 & x_2 = 0,\ x_1 = 1 \\ 0 & x_2 = 0,\ x_1 = 0,\ x_3 = 0 \\ 1 & x_2 = 0,\ x_1 = 0,\ x_3 = 1. \end{cases}$$

Example 8. *According to this definition, the Boolean function $f(x_1, \ldots, x_n) = 0$ is not a nested canalyzing function as the last condition in Equation 2.1 is not satisfied. Note that constant functions are sometimes considered to be trivially canalzying, as in [43, 18].*

In [36], Boolean networks constructed using nested canalyzing functions were shown to be more stable than general Boolean networks in that they are less sensitive to perturbations in the function inputs. In [35], the authors conclude that the dynamics of such networks lie in the frozen phase, which is characterized by small limit cycles. While the results of Peixoto [51] and in Chapter 3 indicate that some of these claims may be due to a pararameterization of the functions, we still believe that these networks are more stable on average than networks of general Boolean functions. Further results regarding stability properties of canalyzing and nested canalyzing functions will be discussed in Chapter 3.

In [31], Jarrah et al. show that the nested canalyzing functions are precisely the

so-called unate cascade functions. Unate cascade functions are the unique class of Boolean functions that have the smallest average path lengths on their binary decision diagrams, and therefore may be evaluated more quickly on average than any other class of Boolean functions [6]. While the relationship between average path length and network stability has not been determined, we believe that this property may be indicative of some type of network stability for NCF systems.

Note that any Boolean function may be represented as a polynomial over $\mathbb{F}_2$ and vice versa. In fact, the ring of Boolean functions in n variables is isomorphic to the polynomial ring $\mathbb{F}_2[x_1, \ldots, x_n]/\langle x_i^2 - x_i : \ 1 \leq i \leq n \rangle$ [31]. In the following examples, some Boolean functions may be presented in their polynomial form for simplicity.

2.2 Simulation Design

In order to compare the stability of networks comprised of nested canalyzing functions with those of general Boolean networks, we performed simulations to compare properties of the state space graphs of each. We developed two measures for determining network stability based on the state space graph, the average cycle length and the average height. The *average cycle length* (ACL) is found by taking the sum over all cycles in the state space of the number of nodes in each cycle, and dividing this by the number of cycles in the graph. We believe that since networks of nested canalyzing functions are believed to lie in a more stable regime, they should have smaller ACLs than networks comprised of general Boolean functions.

Example 9. *The state space graph for the general Boolean network $F = (f_1, f_2, f_3)$, where*

$$\begin{aligned} f_1 &= x_1x_2 + x_3 + x_1x_2x_3 \\ f_2 &= x_1 + x_3 + x_2x_3 + x_1x_2x_3 \\ f_3 &= 1 + x_1 + x_2 + x_2x_3 + x_1x_2x_3 \end{aligned}$$

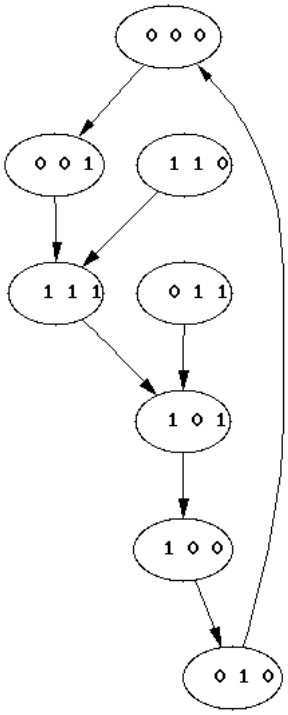

Figure 2.1: State space graph for Example 9 [38]

is given in Figure 2.1. This graph shows a large cycle in the state space, which is not thought to be typical for many biological systems.

The average height of a network measures the average number of discrete time steps until a system converges. We say that the height of a node is the length of the path from the node to a limit cycle, where nodes that are part of a limit cycle have height zero. The *average height* is then the mean of the heights of all nodes in the state space graph. As NCFs may be evaluated more quickly than any other class of Boolean functions, we believe that networks of NCFs should converge more quickly than general Boolean networks and should therefore have smaller average heights.

Example 10. *The state space graph for the general Boolean network* $F = (f_1, f_2, f_3)$*, where*

$$f_1 \quad = \quad 1 + x_1 x_2$$

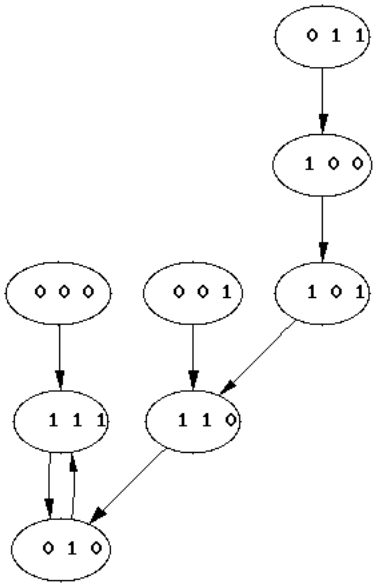

Figure 2.2: State space graph for Example 10 [38]

$$\begin{aligned} f_2 &= 1 + x_1 + x_1x_2 + x_1x_3 + x_2x_3 \\ f_3 &= 1 + x_1x_2 + x_3 + x_1x_2x_3 \end{aligned}$$

is given in Figure 2.2. The long path in this graph gives rise to a large average height for the network, which is not thought to indicate a stable system.

Example 11. *The state space graph for the NCF network* $F = (f_1, f_2, f_3)$*, where*

$$\begin{aligned} f_1 &= \neg x_2 \vee (\neg x_1 \vee \neg x_3) \\ f_2 &= x_2 \wedge (\neg x_1 \wedge \neg x_3) \\ f_3 &= x_3 \wedge (x_2 \wedge \neg x_1) \end{aligned}$$

is given in Figure 2.3. This system has both a small average cycle length and a small average height, both of which are thought to be biologically relevant properties of NCF systems.

We used simulations to compare the average cycle lengths and average heights of

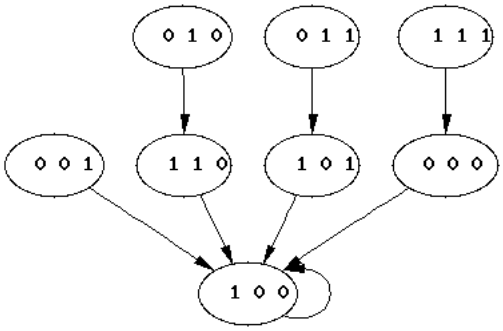

Figure 2.3: State space graph for Example 11 [38]

NCF networks to those of random Boolean networks. Given the number of variables (n) and the number of trials (t), we collected statistics about the dynamics of t random systems of functions in n variables. We generated t systems for both the general Boolean and NCF cases, and ran simulations in C++ for $n = 3, 4, \ldots, 10$. For each trial, we generated a system of n functions in n variables. We evaluated the functions at each possible state to determine state space graph and stored the results. We then recovered the heights of the nodes and the cycle lengths using one of a variety of standard graph algorithms, for instance, a depth first search, and then computed the average cycle length and average height for the entire network. After repeating this procedure for all t trials, we found the mean and standard deviation for ACL and average height over all of the trials. As simply finding the state space graph for a network is $\mathcal{O}(2^n)$, the running time for our simulation algorithm is exponential, requiring us to restrict t as n increases in the interest of time. Obtaining properties of network dynamics without enumerating the entire state space is an ongoing area of research, as in [8, 30].

The C++ header files, fncf.h and fpoly.h, for generating random nested canalyzing functions and Boolean functions, respectively, are provided in Appendix A. Nested canalyzing functions are represented by a vector of canalyzing inputs, **a**, a vector of canalzying outputs, **b**, and a vector storing a permutation of the variables, σ. The vectors **a** and **b** are determined by a random bit generator, while σ is found using the permute function also

located in the file fncf.h. Note that if $b_i = b_{i-1}$, then interchanging σ_i and σ_{i-1} results in a different representation of the same function, thus leading to bias in our function selection. Therefore, we use the check function to ensure that $\sigma(i-1) < \sigma(i)$ whenever $b_i = b_{i-1}$. If this check fails for some i, then we generate new $\mathbf{a}$ and $\mathbf{b}$ vectors until they pass the check. In practice, this check process does not noticeably impact the running time of the algorithm. While we were unable to prove that this procedure gives us a uniform sampling of the nested canalyzing functions, we were able to test it against known NCF counts and properties [48], strongly indicating that our function generator is indeed random.

In the header file fpoly.h, Boolean functions are represented as polynomials via a single vector, *termvec*, constructed using a random bit generator. This vector stores the terms with coefficient one as integers. The binary representation of this integer corresponds to the exponents on the variables for that term. For example, for $n = 3$, if $termvec = [0, 2, 6]$, then our function has three nonzero terms. The binary representation of 0, 000, gives the term $x_3^0 x_2^0 x_1^0 = 1$. The binary representation of 2, 010, gives $x_3^0 x_2^1 x_1^0 = x_2$. Finally the binary representation of 6, 110, gives $x_3^1 x_2^1 x_1^0 = x_3 x_2$, so our function (represented as a polynomial over $\mathbb{F}_2$) is $1 + x_2 + x_2 x_3$. Note that for a Boolean function in n variables generated via this procedure, each of the 2^n possible terms is included in *termvec* with probability $\frac{1}{2}$. Hence, each of the 2^{2^n} possible Boolean functions is selected with probability $\left(\frac{1}{2}\right)^{2^n} = \frac{1}{2^{2^n}}$, and so our method is unbiased, assuming a truly random bit generator.

In [31], the authors provide a count for the number of NCFs in n variables. This count is given by $2 \cdot E(n)$, where

$$E(1) = 1, E(2) = 4,$$

and

$$E(n) = \sum_{r=2}^{n-1} \binom{n}{r-1} 2^{r-1} E(n-r+1) + 2^n.$$

Using this count, we determined that the probability P_n that a randomly generated Boolean

n	t	Bool Mean	NCF Mean	Bool St Dev	NCF St Dev
3	100000	1.9683	1.8003	1.0651	0.9313
4	100000	2.4866	1.9277	1.4934	1.0837
5	100000	3.1408	2.0787	1.9998	1.2386
6	100000	3.9897	2.2165	2.6721	1.3856
7	100000	5.1145	2.3525	3.4972	1.5410
8	10000	6.6044	2.4813	4.6545	1.6742
9	1000	8.6536	2.5685	6.2478	1.7138
10	100	10.8998	2.6231	6.9127	1.9040

Table 2.1: Simulation results: average cycle lengths

network is actually a NCF system is

$$P_n = \left(\frac{2E(n)}{2^{(2^n)}}\right)^n.$$

Therefore, $P_3 \approx 1.56 \cdot 10^{-2}$, $P_4 \approx 1.59 \cdot 10^{-8}$, $P_5 \approx 9.26 \cdot 10^{-29}$, $P_6 \approx 9.83 \cdot 10^{-85}$, etc. Since these probabilities rapidly approach zero, we did not check whether a randomly generated Boolean network system is an NCF system during our simulations.

2.3 Results

The results of our simulations are summarized in the following tables and figures. Table 2.1 records the means and standard deviations of the average cycle lengths for networks comprised of general Boolean functions and NCF networks. The mean average cycle lengths for both the Boolean and NCF cases are plotted in Figure 2.4. Likewise, the summary statistics for the heights of our networks are recorded in Table 2.2. Figure 2.5 compares the average network heights for networks in $n = 3, 4, \ldots, 10$ variables.

Using these results, we performed hypothesis tests for each value of n to determine whether the mean ACLs and heights for general Boolean networks are significantly greater than for those of NCFs. For all cases except average heights where $n = 3$, the p-values for these tests were less than 0.0001, indicating that our results are statistically significant.

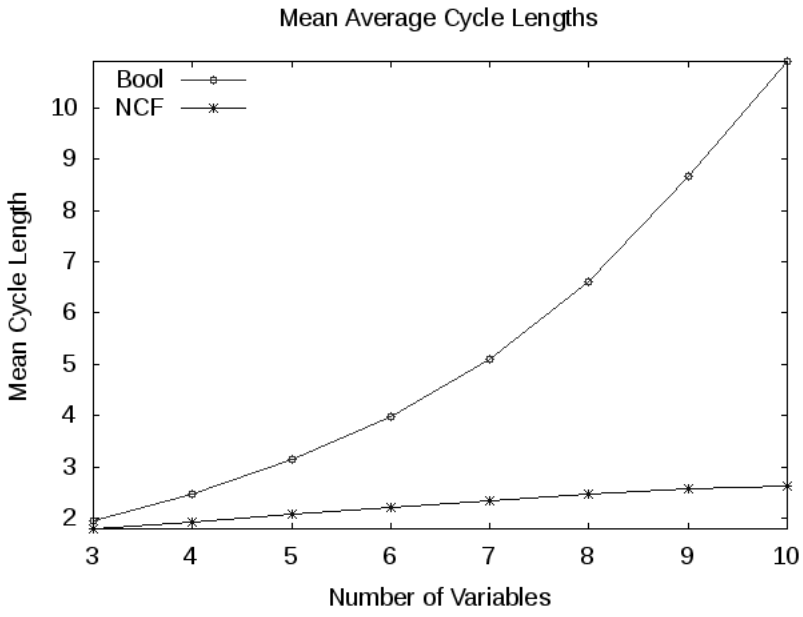

Figure 2.4: Average cycle lengths for Boolean functions and NCFs

n	t	Bool Mean	NCF Mean	Bool St Dev	NCF St Dev
3	100000	1.1176	1.1225	0.6613	0.5653
4	100000	1.8503	1.6463	0.9823	0.6559
5	100000	2.8864	2.1236	1.4348	0.7694
6	100000	4.3610	2.5824	2.0850	0.8929
7	100000	6.4124	3.0180	2.9775	1.0206
8	10000	9.3551	3.4615	4.2908	1.1668
9	1000	13.3790	3.8806	6.0056	1.3515
10	100	18.8491	4.1838	9.1690	1.6476

Table 2.2: Simulation results: average heights

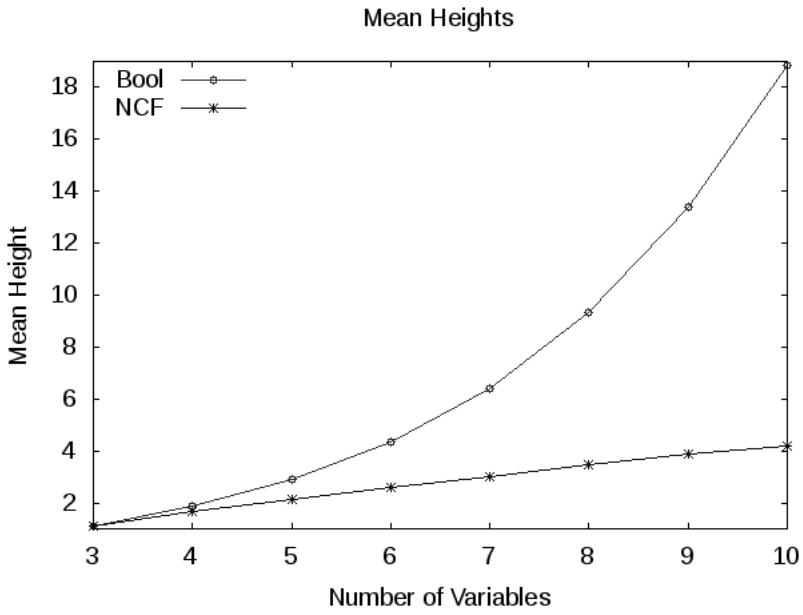

Figure 2.5: Average heights for Boolean functions and NCFs

2.4 Conclusions

Based on our simulation results, we can conclude that the ACLs and average heights for networks of general Boolean functions are significantly larger than those of NCF networks in most of the cases we considered. These findings indicate that NCFs are more stable than typical Boolean functions in the sense that they converge to smaller limit cycles and they converge more quickly on average. Both of these findings seem to indicate that NCFs possess special stability properties, making them appropriate models of biological systems.

The only case for which we did not see statistically significant results was the average heights for $n = 3$. This result could be influenced by a number of factors. First of all, our state space graph consists of only $2^3 = 8$ states, so the graphs may be too small to detect a significant difference in heights. Also, recall that P_3, the probability that a randomly generated network in three variables is an NCF network, is approximately $1.56 \cdot 10^{-2}$. Since NCF networks are much less rare for $n = 3$ than larger networks, the presence of NCF systems by chance in our general Boolean trials is likely influencing our results.

The study of other stability and state space graph properties could lead to further discoveries regarding the dynamics of nested canalyzing functions. For instance, counting the number of connected components or limit cycles in the state space graph could yield significant results. Another gage of function utility may be through taking the maximums of these measures instead of the averages. These observations, in conjunction with previously discovered properties of nested canalyzing functions, indicate that NCFs are a promising class of functions for the modeling of biological systems.

Chapter 3

Nested Canalyzing Depth and Network Stability

Submitted as a paper in 2011 with Elena Dimitrova and Matthew Macauley

3.1 Introduction

A large influx of biological data on the cellular level has necessitated the development of innovative techniques for modeling the underlying networks that regulate cell activities. Several discrete approaches have been proposed, such as Boolean networks [33], logical models [53], and Petri nets [21]. In particular, Boolean networks have emerged as popular models for gene regulatory networks [2, 42]. However, not all Boolean functions accurately reflect the behavior of biological systems, and it is imperative to recognize classes of functions with biologically relevant properties. One such notable class is the canalyzing functions, introduced by Kauffman [33], whose behavior mirrors biological properties described by Waddington [59]. The dynamics of Boolean networks constructed using these functions are of great interest when determining their modeling potential. Random Boolean networks constructed using such functions have been shown to be more stable than networks using general Boolean functions, in the sense that they are insensitive to small perturba-

tions [35]. Karlssona and Hörnquist [32] explore the relationship between the proportion of canalyzing functions and network dynamics. In [35], the authors further expand the canalyzation concept and introduce the class of nested canalyzing functions (NCFs). In [36], networks of NCFs are shown to exhibit stable dynamics. Also, Nikolajewa, et al. [48] divide NCFs into equivalence classes based on their representation and show how the network dynamics are influenced by choice of equivalence class.

Nested canalyzing functions have a very restrictive structure and become increasingly sparse as the number of input variables increases [31]. Also, it is possible that not all variables exhibit canalzying behavior. For instance, transcription factors in gene regulatory networks likely display canalyzing behavior, while other proteins do not. Thus, situations are certain to arise in which nested canalyzing functions do not fully capture the dynamics of biological systems. For instance, the function in Example 13 cannot be represented as an NCF regardless of the variable order. Hence, it is possible that NCFs may not fit a given data set, and so a relaxation of the nested canalyzing function is necessary.

In this chapter, we further explore canalyzation by analyzing functions that retain a partially nested canalyzing structure. We quantify the degree to which a function exhibits this canalyzing structure by a quantity we call the *nested canalyzing depth*. Functions of depth d generalize the nested canalyzing functions, because NCFs are the special case of when $d = k$, where k is the number of Boolean variables. In Section 3.3, we demonstrate notable properties of these partially nested canalyzing functions, and show that their representation is unique. This leads to a theorem about the structure of functions of depth d, which generalizes a result in [31] about NCFs. In Section 3.4, we compute the expected activities and sensitivities of functions given their canalyzing depths, which are extensions of results of Shmulevich and Kauffman [56] about activities and sensitivities of canalyzing functions. We prove that as canalyzing depth increases, functions become less sensitive to perturbations in the input; however, the marginal benefit incurred by adding further canalyzing variables sharply decreases. As a result, functions of larger depth provide an improvement in sensitivity over general canalyzing functions, but imposing a fully nested

canalyzing structure provides little benefit over functions of sufficient canalyzing depth. Finally, in Section 3.5, we use Derrida plots to show that dynamics of networks constructed using more structured functions rapidly approach the well-known critical regime, whereas networks with functions of relatively few nested canalyzing variables remain in the chaotic phase. This is in contrast to the findings of Kauffman et al. [36], but in agreement with recent work of Peixoto [51], and it further supports the biological utility of certain canalyzing functions.

3.2 Nested Canalyzing Depth

A Boolean function $f(x) = f(x_1, \ldots, x_k)$ is *canalyzing* if it has a variable x_i for which some input $x_i = a_i$ implies $f(x) = b_i$ for some $b_i \in \{0, 1\}$. In this case, x_i is a *canalyzing variable*, the input a_i is its *canalyzing value*, and the output value b_i when $x_i = a_i$ is the corresponding *canalyzed value*. Note that if f is constant, then every variable is trivially canalyzing.

If a canalyzing variable x_i does not receive its canalyzing input a_i, then the output is some function $g_i(\hat{x}_i)$, where $\hat{x}_i = (x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_k)$. If g_i is constant, x_i is called a *terminal canalyzing variable* of f. Note that for each $i \neq j$, x_j is then trivially canalyzing in g_i.

If g_i is not constant, we ask whether it too is canalyzing. If so, there is a canalyzing variable x_j with canalyzing input a_j, and when $x_j \neq a_j$, the output of f is a function $g_{ij}(\hat{x}_{ij})$, which may or may not be canalyzing. Here, $\hat{x}_{ij}$ denotes x with both x_i and x_j omitted. Eventually, this process will terminate when the function g is either constant or no longer canalyzing.

Definition 7. *Let $f(x_1, \ldots, x_k)$ be a Boolean function. Suppose that for a permutation*

$\sigma \in S_k$, *some* $d \in \mathbb{N}$, $d > 0$ *and a Boolean function* $g(x_{\sigma(d+1)}, \ldots, x_{\sigma(k)})$,

$$f = \begin{cases} b_1 & x_{\sigma(1)} = a_1 \\ b_2 & x_{\sigma(1)} \neq a_1,\ x_{\sigma(2)} = a_2 \\ b_3 & x_{\sigma(1)} \neq a_1,\ x_{\sigma(2)} \neq a_2,\ x_{\sigma(3)} = a_3 \\ \vdots & \vdots \\ b_d & x_{\sigma(1)} \neq a_1, \ldots,\ x_{\sigma(d-1)} \neq a_{d-1},\ x_{\sigma(d)} = a_d \\ g & x_{\sigma(1)} \neq a_1, \ldots,\ x_{\sigma(d)} \neq a_d \end{cases} \tag{3.1}$$

where either $x_{\sigma(d)}$ *is a terminal canalyzing variable (and hence* g *is constant), or* g *is non-constant and none of the variables* $x_{\sigma(d+1)}, \ldots, x_{\sigma(k)}$ *are canalyzing in* g*. Then* f *is said to be a* partially nested canalyzing function*. The integer* d *is called the* active canalyzing depth *of* f*, and the (full) nested canalyzing* depth *of* f *is* d *if* g *is non-constant, and* k *otherwise. The sequence* $x_{\sigma(1)}, \ldots, x_{\sigma(d)}$ *is called a* canalyzing sequence *for* f*.*

If we speak of simply the "canalyzing depth" or "depth" of a function, we are referring to the full nested canalyzing depth. In the next section, we will show that the depth is well-defined, i.e., that it does not depend on the choice of $\sigma \in S_k$. The class of *nested canalyzing functions* (NCFs) [31, 35] are precisely those with active depth k. A constant function (all 2^k entries in the truth table are the same) is not an NCF by the classical definition, but changing a single value in the truth table suddenly makes it nested canalyzing. In our set-up, both of these functions have full depth k. Note that constant functions have active depth 0. For completeness, we will say that a non-canalyzing function has canalyzing depth 0.

Canalyzing and nested canalyzing functions have been used in gene network models because they possess biologically relevant features [59]. For example, in a gene regulatory network, a collection of k genes that affect the expression level of a particular gene can be modeled with a k-variable Boolean function. While it is believed that some of these relationships are canalyzing (e.g., if A is expressed, then B is not expressed, regardless of

the states of the other genes), it is unreasonable to expect that all relevant genes will act in a nested canalyzing manner. Thus, when reverse engineering a biological network with partial data, the rigid NCF structure is restrictive and likely inappropriate to model the behavior of the system. Also, the number of NCFs becomes rapidly sparse in the number of Boolean functions as k increases. For instance, the proportion of NCFs in 6 variables is on the order of 10^{-15} [31]. Because of this sparsity, it is unlikely that a nested canalyzing function fits a give data set. We will show why functions with less than full canalyzing depth exhibit nearly identical key features as NCFs with regards to activities, sensitivities, and stability, promoting their potential use in biological models.

Example 12. *Let $f(x_1, x_2, x_3) = x_2 \vee (\neg x_1 \wedge x_3)$. Then f is canalyzing in x_2 with canalyzing input $a_1 = 1$ and canalyzed output $b_1 = 1$. Moreover, f can be expressed as follows:*

$$f(x_1, x_2, x_3) = \begin{cases} 1 & \text{if } x_2 = 1 \\ 0 & \text{if } x_2 \neq 1,\ x_1 = 1 \\ 0 & \text{if } x_2 \neq 1,\ x_1 \neq 1,\ x_3 = 0 \\ 1 & \text{if } x_2 \neq 1,\ x_1 \neq 1,\ x_3 \neq 0 \end{cases}.$$

Thus, f has full and active depth 3, and so it is nested canalyzing in the variable order $\sigma = (2, 1, 3)$.

Example 13. *Let $f(x_1, x_2, x_3, x_4) = x_2 \wedge (\neg x_1 (\wedge (x_3\ XOR\ x_4)))$. Then x_2 and x_1 are nested canalyzing variables with $a_1 = a_2 = 1$, $b_1 = 1$, and $b_2 = 0$. Also, $g(x_1, x_4) = x_1\ XOR\ x_4$ is not canalyzing in either variable, so f is a PNCF of depth 2.*

3.3 Properties of Partially Nested Canalyzing Functions

Proposition 1. *Let $f(x)$ be a k-variable Boolean function. Then*

(i) If $x_i = a_i$ implies $f(x) = b_i$, and $x_i \neq a_i$ leaves $g_i(\hat{x}_i)$, then at least half of the truth table values of f must be b_i.

(ii) If exactly half the truth table values of f are b_i, then either x_i is terminally canalyzing, or f is a non-canalyzing function.

Proof. The statement in (i) follows because $x_i = a_i$ for exactly half of the input values in the truth table. The corresponding output value must b_i for at least these inputs. To show (ii), suppose that f is canalyzing, and $x_i = a_i$ implies $f(x) = b_i$. By (i), $x_i \neq a_i$ implies $f(x) = \neg b_i$. Therefore, $g(\hat{x}_i)$ is constant and x_i is terminally canalyzing. □

The canalyzing depth of a function can be computed in a divide-and-conquer manner described in Algorithm 1. The algorithm scans the truth table for a canalyzing variable, and upon finding one, removes the columns for which the canalyzing variable takes the canalyzing input value. This is repeated until no more canalyzing variables are present or a constant function remains. Proposition 1 and the structure of the truth table imply that if there is a tie for b, then there is a terminally canalzying variable or there are no canalyzing variables. Therefore, it is not necessary to test both b and $\neg b$ as possible canalyzed values. In the execution of the algorithm, we set a flag whenever a tie for canalyzed value arises.

Algorithm 1.

1. *Set $d = 0$. For $i = 1 \ldots k-1$:*

   (a) *Set $b = 0$, flag $= 0$. Let ℓ be the number of ones in the truth table.*

   - *If $\ell == 2^{k-i+1}$ return k. // Constant function remains*
   - *If $\ell == 2^{k-i}$, set flag $= 1$. // Tie in output value*
   - *If $\ell > 2^{k-i}$, $b = 1$.*

   (b) *For remaining $k - i + 1$ variables in truth table:*

   i. *Let x be the number of input ones and y the number of input zeros that give output b.*

   ii. - *If $x == 2^{k-i}$, the current variable is canalyzing with input 1 and output b. Remove canalyzing rows and current variable from truth table and break out of loop.*

- *If $y == 2^{k-i}$, the current variable is canalyzing with input 0 and output b. Remove canalyzing rows and current variable from truth table and break out of loop.*

(c) If no variables were found to be canalyzing, return d; else d++.

(d) If flag $== 1$, return k. // Constant function remains

2. Return k.

Note that it takes exponential time simply to view the entire truth table of f; however, the algorithm is linear in the size of the table. Indeed, the i^{th} step of Algorithm 1 takes $(k-i)\cdot 2^{k-i}$ steps, and so the running time is

$$\sum_{i=1}^{k}(k-i)2^{k-1} \leq k\cdot 2^k\left(1+\frac{1}{2}+\frac{1}{4}+\ldots\right)=\mathcal{O}\left(k\cdot 2^k\right).$$

We can use Algorithm 1 to show that our definition of canalyzing depth is well-defined. First, we need a result about the struture of functions with a given canalyzing depth. This is a generalization of a theorem in [31] on nested canalyzing functions.

Theorem 1. *Let $y_i = x_{\sigma(i)} + a_i + b_i$, $1 \leq i \leq d$ and let*

$$f(x_1,\ldots,x_k) = y_1 \lozenge_1 (y_2 \lozenge_2 (\ldots (y_d \lozenge_d g(x_{\sigma(d+1)},\ldots,x_{\sigma(k)}))\ldots)), \tag{3.2}$$

where

$$\lozenge_i = \begin{cases} \vee & if\ b_i = 1 \\ \wedge & if\ b_i = 0 \end{cases},$$

$a_i, b_i \in \{0,1\}$ for $1 \leq i \leq d$, and

(i) None of the variables $x_{\sigma(d+1)},\ldots,x_{\sigma(k)}$ are canalyzing in g, or

(ii) g is a constant function.

Then f has canalyzing depth d, with canalyzing sequence $x_{\sigma(1)}, \ldots, x_{\sigma(d)}$. These variables have canalyzing values $a_1, \ldots, a_d$ and canalyzed values $b_1, \ldots, b_d$. Furthermore, any function of canalyzing depth d can be represented in this form.

The aforementioned result in [31] is the special case of Theorem 1 when f is nested canalyzing. In this case,

$$f(x_1, \ldots, x_k) = y_1 \Diamond_1 (y_2 \Diamond_2 (\cdots (y_{k-1} \Diamond_{k-1} y_k) \cdots)).$$

Proposition 1 and our previous observations indicate that in case of a tie in potential canalyzed values, we cannot make a "wrong" choice for b in the execution of Algorithm 1. Hence, to show that the depth is unique, it suffices to show that if there are multiple canalyzing variables at a given iteration, the depth does not depend on our choice of canalyzing variable.

Proposition 2. *The nested canalyzing depth computed using Algorithm 1 yields a unique answer.*

This result follows from symmetry and structure of the truth table, as well as Proposition 1. It is now easy to see that the nested canalyzing structure introduced in Equation 3.1 is well-defined since the remaining function g is unique.

3.4 Activities and Sensitivities

In this section we compute the expected activities and sensitivities of functions based on their canalyzing depth, and in the next section we will tie these results to the stability of Boolean networks based on the canalyzing depth of the individual functions. Let $\mathbf{x} \in \{0,1\}^k$, and write $\mathbf{x}^{j,i} = (x_1, \ldots, x_{j-1}, i, x_{j+1}, \ldots, x_k)$ and let $\oplus$ be the XOR function. The partial derivative of $f(x_1, \ldots, x_k)$ with respect to x_j is

$$\frac{\partial f(\mathbf{x})}{\partial x_j} = f(\mathbf{x}^{j,0}) \oplus f(\mathbf{x}^{j,1}).$$

The activity (or influence) of a variable x_j in f is

$$\alpha_j^f(\mathbf{x}) = \frac{1}{2^k} \sum_{\mathbf{x} \in \{0,1\}^k} \frac{\partial f(\mathbf{x})}{\partial x_j} \tag{3.3}$$

and the sensitivity of f is defined by

$$s^f(\mathbf{x}) = \sum_{i=1}^{k} \chi\left[f(\mathbf{x} \oplus e_i) \neq f(\mathbf{x})\right],$$

where e_i is the i^{th} unit vector and χ is an indicator function. The activity α_j^f quantifies how often toggling the j^{th} bit of $\mathbf{x}$ toggles the output of f, and the sensitivity $s^f(\mathbf{x})$ measures the number of ways that toggling a bit of $\mathbf{x}$ toggles the output of f. The average sensitivity of f is the expected value of $s^f(\mathbf{x})$ taken uniformly over all $\mathbf{x} \in \{0,1\}^k$, i.e.,

$$s^f = E\left[s^f(\mathbf{x})\right] = \sum_{i=1}^{k} \alpha_i^f. \tag{3.4}$$

In [56], Shmulevich and Kauffman show that a random unbiased Boolean function in k variables has average sensitivity $\frac{k}{2}$. Also, they prove that for an unbiased canalyzing function (i.e., depth at least 1) with canalyzing variable x_1, the expected activities of $(x_1, \ldots, x_k)$ are given by

$$E\left[\alpha^f\right] = \left(\frac{1}{2}, \frac{1}{4}, \frac{1}{4}, \ldots, \frac{1}{4}\right), \tag{3.5}$$

and hence the average sensitivity is $s^f = \frac{k+1}{4}$. The following theorem extends this to functions of arbitrary canalyzing depth.

Theorem 2. *Let f be a Boolean function in k variables with nested canalyzing depth at least d. Renumbering the variables if necessary, assume that $x_1, \ldots, x_d$ is a canalyzing sequence. Then, if we assume a uniform distribution on the function inputs, the expected activities of the variables $(x_1, \ldots, x_k)$ are given by*

$$E\left[\alpha^f\right] = \left(\frac{1}{2}, \frac{1}{4}, \ldots, \frac{1}{2^d}, \frac{1}{2^{d+1}}, \ldots, \frac{1}{2^{d+1}}\right). \tag{3.6}$$

Furthermore, the expected sensitivity of f is

$$E\left[s^f\right] = \frac{k-d}{2^{d+1}} + \sum_{i=1}^{d} \frac{1}{2^i} = \frac{k-d}{2^{d+1}} + 1 - \frac{1}{2^d}. \tag{3.7}$$

Proof. Since we are assuming a uniform distribution on the function inputs, for any variable x_j, $1 \leq j \leq k$, we can think of the activity of x_j as the probability that changing the input to the x_j changes the function output. That is,

$$\begin{aligned} \alpha_j^f(\mathbf{x}) &= \frac{1}{2^k} \sum_{\mathbf{x} \in \{0,1\}^k} \frac{\partial f(\mathbf{x})}{\partial x_j} \\ &= P(f(\mathbf{x} \oplus e_j) \neq f(\mathbf{x})). \end{aligned}$$

Now, if x_j is a canalyzing variable, we know by Equation 3.1 that if at least one of $x_1, \ldots, x_{j-1}$ gets its canalyzing input, the input to x_j cannot affect the function output and this probability is 0. Hence, we have

$$\begin{aligned} \alpha_j^f &= P(f(\mathbf{x} \oplus e_j) \neq f(\mathbf{x})) \\ &= P(f(\mathbf{x} \oplus e_j) \neq f(\mathbf{x}) | x_1 \neq a_1, \ldots, x_{j-1} \neq a_{j-1}) P(x_1 \neq a_1, \ldots, x_{j-1} \neq a_{j-1}). \end{aligned}$$

Since each canalyzing variable receives its canalyzing input with probability $\frac{1}{2}$,

$$P(x_1 \neq a_1, \ldots, x_{j-1} \neq a_{j-1}) = \left(\frac{1}{2}\right)^{j-1}.$$

Also, since f is a random, unbiased function,

$$P(f(\mathbf{x} \oplus e_j) \neq f(\mathbf{x}) | x_1 \neq a_1, \ldots, x_{j-1} \neq a_{j-1}) = \frac{1}{2}.$$

Therefore,

$$\alpha_j^f = \frac{1}{2} \cdot \frac{1}{2^{j-1}} = \frac{1}{2^j}.$$

Alternatively, if x_j is a non-canalyzing variable, the input to x_j is only relevant

when none of the canalyzing variables $x_1, \ldots x_d$ get their canalyzing inputs. Using a similar argument as above, we see that

$$\begin{aligned}
\alpha_j^f &= P(f(\mathbf{x} \oplus e_j) \neq f(\mathbf{x})) \\
&= P(f(\mathbf{x} \oplus e_j) \neq f(\mathbf{x}) | x_1 \neq a_1, \ldots, x_d \neq a_d) P(x_1 \neq a_1, \ldots, x_d \neq a_d) \\
&= \frac{1}{2} \cdot \frac{1}{2^d} \\
&= \frac{1}{2^{d+1}}.
\end{aligned}$$

Equation 3.7 now follows from Equation 3.4. □

Note that an alternative proof for this theorem follows via induction on d, with Equation 3.5 as a base case, following an argument similar to that in [56].

By Theorem 2, the average sensitivity of a function decreases as the depth increases. However, the differences in sensitivity become increasingly smaller, and are precisely

$$\begin{aligned}
E[s^{f_d}] - E[s^{f_{d+1}}] &= 1 - \tfrac{1}{2^d} + \tfrac{k-d}{2^{d+1}} - 1 + \tfrac{1}{2^{d+1}} - \tfrac{k-d-1}{2^{d+2}} \\
&= \tfrac{k-d-1}{2^{d+2}} \geq 0\,, \quad \text{when } k - d \geq 1\,.
\end{aligned}$$

Observe that this quantity rapidly goes to zero, and so each subsequent canalyzing variable has a much smaller impact on the sensitivity. Thus, the difference in sensitivity between fully nested canalyzing functions and partially nested canalyzing functions of sufficient depth is very slight. For example, Tables 3.1 and 3.2 give the expected sensitivities for PNCFs with $k = 6$ and $k = 12$, respectively.

Table 3.1: Expected sensitivities for PNCFs in 6 variables of various depths

d	0	1	2	3	4	5	6
$E\left[s^{f_d}\right]$	3.0000	1.7500	1.2500	1.0625	1.0000	0.9844	0.9844

Table 3.2: Expected sensitivities for PNCFs in 12 variables of various depths

d	0	2	4	6	8	10	12
$E\left[s^{f_d}\right]$	6.0000	2.0000	1.1875	1.0313	1.0039	1.0000	0.9998

3.5 Stability and Criticality vs. Canalyzing Depth

Boolean networks created using classes of functions with a lower sensitivity have been shown to be more dynamically ordered than those with a higher sensitivity [56]. This stability is an important feature of biologically relevant functions, and so it is essential to determining the utility of such functions as biological models. In order to quantify the extent to which functions with larger depth (and hence smaller sensitivity) result in more dynamically stable Boolean networks, we constructed random Boolean networks composed of PNCFs of varying depth. We used the annealed approximation mean-field theory due to [14] and *Derrida curves* to display the results. The curves are defined as follows. Let $\mathbf{x}^1(t)$ and $\mathbf{x}^2(t)$ be two states in a random Boolean network, and define $\rho(t)$ to be the normalized Hamming distance, i.e., $\rho(t) = \frac{1}{n} \cdot ||\mathbf{x}^1(t) - \mathbf{x}^2(t)||_1$, where $||\cdot||_1$ is the standard ℓ^1 metric. The Derrida curve is a plot of $\rho(t+1)$ versus $\rho(t)$ averaged uniformly over different states and networks. If the curve for small values of $\rho(t)$ lies below the line $y = x$, then small perturbations are likely to die out, and the network is said to be in the *frozen* phase. The phase spaces of frozen networks consist of many fixed points and small attractor cycles. If the curve lies above the line $y = x$, then small perturbations generally propagate throughout the network, and the network is said to be in the *chaotic* phase, characterized by long attractor cycles. The boundary threshold between these two is the well-known *critical* phase [18]. It has been recently suggested [4, 49, 50, 57] that many biological networks tend to lie in the critical phase, as these systems must be stable enough to endure changes to their environment, yet flexible enough to adapt when necessary.

We constructed ensembles of randomly wired networks with $n = 100$ nodes, each with a randomly chosen Boolean function with $k = 12$ variables. We chose the individual functions by sampling uniformly across the class of PNCFs of depth *at least* d, for $d =$

$0, 2, 4, \ldots, 12$. We will refer to such a network as a *depth-d network*. To sample uniformly across all PNCFs of depth at least d, we used a random number generator to select d nested canalyzing variables, and a permutation σ of these variables. We then used a random bit generator to select the canalyzing values $a_1, \ldots, a_d$ and canalyzed values $b_1, \ldots b_d$. We had a potential bias in our function selection arising from the fact that if $\diamond_i = \diamond_{i+1}$ (or equivalently, $b_i = b_{i+1}$, as b_i determines $\diamond_i$), then interchanging $\sigma(i)$ with $\sigma(i+1)$ does not change the function. To eliminate this bias, we only allowed functions where $\sigma(i-1) < \sigma(i)$ whenever $\diamond_{i-1} = \diamond_i$ for $i = 2, \ldots, d-1$. Finally, we used a random bit generator to determine the function in the remaining $k - d$ variables. Our sampling method for creating the random networks is similar to [56]. For each d, we created 25 random Boolean networks using functions of said depth and sampled from each network. Since $\rho(t+1)$ is determined experimentally, we computed it as the sample mean, sampled over the depth-d random networks for each depth. We also constructed Derrida curves using the sampling method described in [34], which generated nearly identical results. The resulting Derrida curves are shown in Figure 3.1.

The Derrida curves corresponding to networks constructed using functions of larger depth show more orderly dynamics than those of smaller depth. This reaffirms the idea in [56] that sensitivity of a function is an indicator of the dynamical stability of networks constructed with these functions. The curves move closer together as the depth increases. For example, the depth-2 networks are much more stable than the depth-0 networks, and networks with functions of depth at least 4 are even more stable; however, the marginal benefit of stability as depth increases drops off sharply – the Derrida curves are nearly identical for networks with functions of depth 4, 6, 8, 10, and 12. This matches our theoretical results of Theorem 2 on expected activities and sensitivities and illustrates how the earlier canalyzing variables have a much greater influence. It also suggests that there is little benefit in imposing the full nested canalyzing structure in network models, as functions with large enough canalyzing depth exhibit very similar stability results without the rigidity of being fully nested canalyzing. Additionally, for small values of $\rho(t)$, the curves quickly

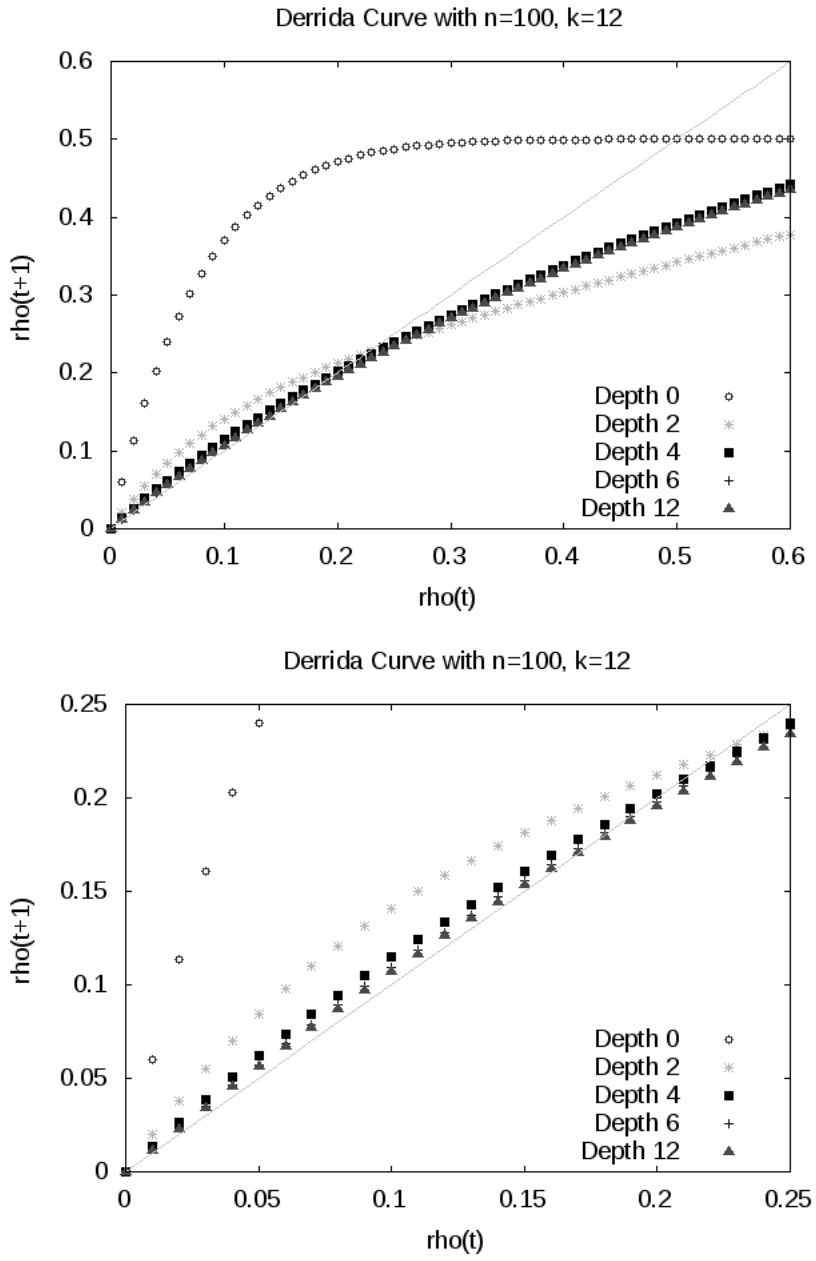

Figure 3.1: Derrida curves for random Boolean networks with $n = 100$ nodes and $k = 12$ inputs per function.

approach the line $y = x$ from above, indicating that these networks rapidly move from the chaotic phase toward the critical phase. This is contrary to the claim of [35] that networks comprised of canalyzing functions are always in the frozen phase, but it is in alignment with recent findings of [51] which also refute Kauffuman's claim, accrediting his results to his choice in parametrization.

3.6 Concluding Remarks

Canalyzing and nested canalyzing functions have been proposed as gene network models because they exhibit biologically relevant properties. While it is reasonable to expect some Boolean models to have functions with some degree of nested canalyzation, fitting biological data to fully nesting canalyzing functions can be at times artificial, and at other times simply incorrect. Our analysis of the depth that a function retains a canalyzing structure elucidates the role of canalyzation in the dynamics of networks over these functions. Our results on the structure of PNCFs generalize known results on NCFs, and our results on the activities and sensitivities of variables in functions of a given depth generalize similar theorems of simple canalyzing functions. Moreover, we saw that in random Boolean networks, the stability increases with canalyzing depth. However, the marginal gain in stability drops off quickly, in that the stability of our networks with functions of depth at least $d = \frac{k}{3}$ were nearly identical to those with full depth $d = k$. Additionally, just a few degrees of canalyzation are necessary to drop the network into the critical regime, in which many real networks are believed to exist. Together, this suggests that using NCFs in biological models for stability reasons is not only at times rather contrived, but simply unnecessary.

Chapter 4

Reverse Engineering with Partially Nested Canalyzing Functions

4.1 Introduction

Modeling the dynamic behavior of biochemical networks has emerged as a fundamental problem in systems biology. Several discrete frameworks have been proposed as network models, including Petri nets [21], Logical models [53], and Boolean networks [2, 42]. In 2004, Laubenbacher and Stigler introduced a revolutionary reverse engineering algorithm for gene regulatory networks using tools and algorithms from computational algebra [40]. Given discretized gene expression data in a discrete time series, the algorithm finds all polynomial dynamical systems satisfying a minimality criterion that fit the data. We will focus on Boolean networks, which are a special case of these polynomial models.

One key deficiency of the Laubenbacher and Stigler algorithm is that any Boolean function may be chosen to be a network model. In order to obtain a more biologically relevant model, we would like to restrict our search to only include Boolean functions whose behavior mimics real biological networks. To this end, Kauffman [33] introduced the canalyzing functions due to their biologically relevant properties. For instance, Boolean networks constructed from canalyzing functions are less sensitive to network perturbations

than random Boolean networks [35, 56]. The nested canalyzing functions, an extension of the canalyzing functions, were introduced in [35] and were shown to be more stable than general Boolean functions in [36]. Also, in Chapter 2, we showed via simulation that networks constructed using canalyzing functions are more stable than general Boolean networks in the sense that they converge more quickly and to smaller limit cycles on average. Hinkelmann and Jarrah [27] designed an algorithm to reverse engineer gene regulatory networks using only nested canalyzing functions. In Chapter 3, we introduced a generalization of the nested canalyzing functions, the partially nested canalyzing functions, characterized by their *depth*, or degree to which they exhibit a nested canalyzing behavior. We showed that as the depth increases, the activities of the variables decreases, as well as the expected sensitivities of the functions. Also, when constructing networks restricted to PNCFs, the Derrida curves for these networks approach the biologically relevant critical regime of dynamics. We observed diminishing returns as function depth increases, implying that fully nested canalyzing functions, whose structure is more rigid than PNCFs, are not necessary to achieve desirable stability properties. In this chapter, we will introduce a method for reverse engineering gene regulatory networks using partially nested canalyzing functions.

4.2 Reverse Engineering Algorithms

A *polynomial dynamical system* (PDS) is a map

$$F = (f_1, \ldots, f_n) : \mathbb{F}^n \to \mathbb{F}^n,$$

where $\mathbb{F}$ is a finite field, typically $\mathbb{F}_p$, n is the number of nodes (genes) in the network, and each f_i is a polynomial in n variables of degree at most $p-1$. We will focus on the Boolean case where $p = 2$. Given a time series of r data points $\mathbf{s}_1, \ldots, \mathbf{s}_r \in \mathbb{F}_2^n$, our goal is to find biologically relevant Boolean networks such that

$$F(\mathbf{s}_i) = (f_1(\mathbf{s}_i), \ldots, f_n(\mathbf{s}_i)) = \mathbf{s}_{i+1},\ 1 \leq i \leq r-1. \tag{4.1}$$

4.2.1 Laubenbacher-Stigler Algorithm

In [40], the authors introduce a reverse engineering algorithm for gene networks using polynomial dynamical systems. Given a time series of data $\mathbf{s}_1, \ldots, \mathbf{s}_r \in \mathbb{F}^n$, their algorithm finds all polynomials $f_1, \ldots, f_n \in \mathbb{F}[x_1, \ldots, x_n]$ such that (4.1) holds. The algorithm consists of three steps:

1. Find an interpolating polynomial $f_i^0 \in \mathbb{F}[x_1, \ldots, x_n]$, for each i, $1 \leq i \leq n$.

2. Compute the ideal I of functions that vanish on the data.

3. Reduce f_i^0 with respect to I to obtain f_i for $i = 1, \ldots, n$.

The interpolating polynomial in Step 1 is analogous to a particular solution and may be computed using one of several common methods. The authors present a method based on the Chinese Remainder Theorem, which they utilize in their applications.

Note that if two interpolating polynomials, say f_i and g_i, exist for some node i, then $f_i(\mathbf{s}_j) = g_i(\mathbf{s}_j) = \mathbf{s}_{j+1}$ for $1 \leq j \leq r-1$. This means that the difference between these functions is 0 on the data. Therefore, to find all functions that fit the data, we must find the set of *vanishing polynomials* on the data, which may be found by computing the *ideal of points*. Observe that for a data point $\mathbf{s}_i = \{s_{i1}, \ldots, s_{in}\}$, the ideal I_i of polynomials that vanish on $\mathbf{s}_i$ is as follows

$$I_i = \langle x_1 - s_{i1}, \ldots, x_n - s_{in} \rangle.$$

Now, the ideal of points I of all functions that vanish on the data is

$$I = \bigcap_{i=1}^{r} I_i.$$

We now have all interpolating functions for each node i, which may be expressed as $f_i^0 + I$. Recall that f_i^0 is analogous to a particular solution, so I may be thought of as the homogeneous solutions.

Finally, each f_i^0 is reduced with respect to I to obtain f_i. This is done by computing a Gröbner basis for I and then finding the normal form for f_i^0 with respect to the Gröbner basis. As a result, we can write $f_i^0 = f_i + g$ for $g \in I$. The authors give two advantages for this method. First, the output does not depend on our selection of an interpolating polynomial in Step 1, as any two interpolating polynomials will have the same reduction modulo I. Second, f_i is minimal with respect to I, and since g vanishes on the data, the state transition function for node i is simply f_i.

The authors state that the complexity of the algorithm is

$$\mathcal{O}\left(n^2r^2\right) + \mathcal{O}\left((r^3+r)(\log p)^2 + r^2n^2\right) + \mathcal{O}\left(n(r-1)2^{cr+r-1}\right),$$

where n is the number of nodes in the network, r is the number of data points in the time series, p is the order of the field, and c is a constant. The first expression is the time it takes to compute interpolating polynomials for all n nodes. The second expression gives the computation time for computing the Gröbner basis for I. In general, the worst-case running time for a Gröbner basis computation is thought to be exponential; however, since I is an ideal of points, this can be reduced using methods described in [1]. The final expression, which is the bottleneck in the algorithm, is the complexity of reducing all n interpolating polynomials modulo I, and is exponential in the number of data points.

This algorithm has two notable drawbacks. First, the Gröbner basis is dependent on the choice of monomial ordering, which results in multiple possible solutions. To remedy this problem, a *Gröbner fan* method has been proposed that finds the most likely model among different monomial orderings [15]. Another considerable disadvantage with this method is that any function in $\mathbb{F}[x_1, \ldots, x_n]$ fitting the data may be selected as a network model. We would like to restrict the model space to only include functions whose behavior reflects that of real biological systems. For instance, as the nested canalzying functions seem to exhibit biologically relevant properties, restricting our search to only include NCFs would potentially lead to more realistic network models. In the following sections, we will

explore reverse engineering methods that only include nested and partially nested canalyzing functions due to their biologically significant stability properties.

4.2.2 NCF Algorithm

Canalyzing functions were introduced by Kauffman [33], which reflect the behavior of biological systems described by Waddingdon [59]. Specifically, Waddington introduced the term "canalization" to describe the ability of a genotype to yield the same phenotype despite variations in the environment. Mathematically speaking, a function $f(x_1, \ldots, x_n)$ is said to be *canalyzing* if there exists a variable x_i, $1 \leq i \leq n$ and values $a, b \in \{0, 1\}$ such that $f(x_1, \ldots, x_{i-1}, a, x_{i+1}, \ldots, x_n) = b$ regardless of the inputs to the other variables. In [35], the authors introduce the nested canalyzing functions (NCFs), which specify how to proceed if the canalyzing variable x_i does not receive its canalyzing input a. The definition and examples of nested canalyzing functions are found in Chapter 2.

We now introduce a characterization of NCFs introduced in [31] that gives rise to a reverse engineering algorithm. The ring of Boolean functions has been shown to be isomorphic to the ring $R = \mathbb{F}_2[x_1, \ldots, x_n]/\langle x_i^2 : 1 \leq i \leq n \rangle$. Also, given a fixed ordering of the monomials in this ring, any element in R may be expressed as a point in $\mathbb{F}_2^{2^n}$ as follows

$$\sum_{S \subseteq [n]} c_S \prod_{i \in S} x_i \leftrightarrow (c_\emptyset, \ldots, c_{[n]}),$$

where $[n] = \{1, 2, \ldots, n\}$ and the c_S values correspond to the monomial coefficients. Now, the representation of the nested canalyzing functions as an algebraic variety follows from the subsequent results of [31].

Definition 8. *Let $\sigma \in S_n$ and for elements i and j of $[n]$, define $<_\sigma$ by $\sigma(i) <_\sigma \sigma(j)$ if and only if $i < j$. For $S \subseteq n$, let r_S^σ be the maximum element of S with respect to σ. Then the* completion of S with respect to σ *is given by $[r_S^\sigma] = \{\sigma(1), \ldots, \sigma(r_S^\sigma)\}$.*

Theorem 3. *Let $f \in R$ and $\sigma \in S_n$. Then f is a nested canalyzing function with respect*

to σ if and only if $c_{[n]} = 1$ and for any $S \subseteq [n]$,

$$c_S = c_{[r_S^\sigma]} \prod_{\sigma(i) \in [r_S^\sigma] \setminus S} c_{[n] \setminus \{\sigma(i)\}}.$$

Now, for a fixed permutation $\sigma \in S_n$, the variety of all nested canalyzing functions with respect to σ is

$$V_\sigma^{\mathrm{ncf}} = \{(c_\emptyset, \ldots, c_{[n]}) \in \mathbb{F}_2^{2^n} : c_{[n]} = 1,\ c_S = c_{[r_S^\sigma]} \prod_{\sigma(i) \in [r_S^\sigma] \setminus S} c_{[n] \setminus \{\sigma(i)\}},\ \text{for } S \subseteq [n]\}.$$

Jarrah et al. also show that the variety of nested canalyzing functions in n variables is

$$V^{\mathrm{ncf}} = \bigcup_\sigma V_\sigma^{\mathrm{ncf}} \text{ [31]}.$$

In [28], the authors show that ideal $\mathbb{I}\left(V_\sigma^{\mathrm{ncf}}\right)$ in the ring $\overline{\mathbb{F}}_2[\{c_S : S \subseteq [n]\}]$ given by

$$\mathbb{I}\left(V_\sigma^{\mathrm{ncf}}\right) = \langle c_{[n]} = 1,\ c_S = c_{[r_S^\sigma]} \prod_{\sigma(i) \in [r_S^\sigma] \setminus S} c_{[n] \setminus \{\sigma(i)\}} : S \subseteq [n] \rangle$$

is a toric ideal. In addition, the ideal of the variety V^{ncf} is

$$\mathbb{I}\left(V^{\mathrm{ncf}}\right) = \bigcap_\sigma \mathbb{I}\left(V_\sigma^{\mathrm{ncf}}\right) \text{ [28]}.$$

Based on these results, Hinkelmann and Jarrah propose the following reverse engineering algorithm using NCFs [27]. Given time series data and a wiring diagram, the algorithm finds all nested canalzying functions that fit the data. If the wiring diagram is unknown, algorithms exist for building one from data itself. In general, finding a wiring diagram that is minimal in the sense that it excludes redundant input variables is NP-hard [37]; however, these computations are more manageable in practice due to the sparsity of most biological networks [29]. A method for finding all minimal wiring diagrams that is independent of the selection of monomial ordering is presented in [29].

Based on the ideal of points, the authors construct an ideal I_D of interpolating polynomials for the data set D. Then the points in $\mathbb{V}(I_D)$ are the models $f + I$ that fit the data, as in [40]. Further details on the construction of I_D may be found in [27]. They compute a Gröbner basis G for I_D and concatenate the generators of G with those of $\mathbb{I}\left(V^{\text{ncf}}\right)$. Finally, they compute the primary decomposition of $G + \mathbb{I}\left(V^{\text{ncf}}\right)$ to acquire the result. While the authors do not explicitly state the complexity of the algorithm, the bottlenecks appear to be the Gröbner basis computation and the primary decomposition.

4.3 PNCF Algorithm

4.3.1 Partially Nested Canalyzing Functions

While the nested canalyzing functions are excellent candidates for modeling biological networks due to their desirable stability properties and convenient ideal representation, situations are certain to arise in which not all genes behave in a nested canalyzing fashion or an NCF simply does not fit the data. For example, consider the sample data in Table 4.1. The expression level of the x_4 output depends on those of x_1, x_2, and x_3. Observe that x_1 is canalyzing with canalyzing input 1 and canalyzed output 0. When this variable does not receive its canalyzing input, the remaining data in x_2 and x_3 is not canalyzing in either variable, thus violating the conditions for an NCF. In this case, it would be necessary to fit a PNCF to the data. Therefore it is imperative to consider a relaxation of the nested canalyzing functions, called the *partially nested canalyzing functions* (PNCFs) whenever the full nested canalyzing conditions do not hold. Recall from Chapter 3 the following definition

Table 4.1: Sample data that is inconsistent with an NCF

x_1	x_2	x_3	x_4 output
1	0	0	0
0	0	1	1
0	0	0	0
0	1	1	0
1	1	0	1
0	1	0	0

and examples of PNCFs.

Definition 9. *Let $f(x_1, \ldots, x_n)$ be a Boolean function. Suppose that for a permutation $\sigma \in S_n$, some $d \in \mathbb{N}$, $d > 0$ and a Boolean function $g(x_{\sigma(d+1)}, \ldots, x_{\sigma(n)})$,*

$$f = \begin{cases} b_1 & x_{\sigma(1)} = a_1 \\ b_2 & x_{\sigma(1)} \neq a_1,\ x_{\sigma(2)} = a_2 \\ b_3 & x_{\sigma(1)} \neq a_1,\ x_{\sigma(2)} \neq a_2,\ x_{\sigma(3)} = a_3 \\ \vdots & \vdots \\ b_d & x_{\sigma(1)} \neq a_1, \ldots,\ x_{\sigma(d-1)} \neq a_{d-1},\ x_{\sigma(d)} = a_d \\ g & x_{\sigma(1)} \neq a_1, \ldots,\ x_{\sigma(d)} \neq a_d \end{cases} \tag{4.2}$$

where either $x_{\sigma(d)}$ is a terminal canalyzing variable (and hence g is constant), or g is non-constant and none of the variables $x_{\sigma(d+1)}, \ldots, x_{\sigma(n)}$ are canalyzing in g. Then f is said to be a partially nested canalyzing function. *The integer d is called the* active canalyzing depth *of f, and the (full) nested canalyzing* depth *of f is d if g is non-constant, and k otherwise. The sequence $x_{\sigma(1)}, \ldots, x_{\sigma(d)}$ is called a* canalyzing sequence *for f.*

Example 14. *Let $f(x_1, x_2, x_3) = x_2 \vee (\neg x_1 \wedge x_3)$. Then f is canalyzing in x_2 with canalyzing input $a_1 = 1$ and canalyzed output $b_1 = 1$. Moreover, f can be expressed as follows:*

$$f(x_1, x_2, x_3) = \begin{cases} 1 & \text{if } x_2 = 1 \\ 0 & \text{if } x_2 \neq 1,\ x_1 = 1 \\ 0 & \text{if } x_2 \neq 1,\ x_1 \neq 1,\ x_3 = 0 \\ 1 & \text{if } x_2 \neq 1,\ x_1 \neq 1,\ x_3 \neq 0 \end{cases}.$$

Thus, f has full and active depth 3, and so it is nested canalyzing in the variable order $\sigma = (2, 1, 3)$.

Example 15. *Let $f(x_1, x_2, x_3, x_4) = x_2 \wedge (\neg x_1 (\wedge (x_3\ XOR\ x_4)))$. Then x_2 and x_1 are nested canalyzing variables with $a_1 = a_2 = 1$, $b_1 = 1$, and $b_2 = 0$. Also, $g(x_1, x_4) = x_1\ XOR\ x_4$ is not canalyzing in either variable, so f is a PNCF of depth 2.*

In Chapter 3, we also introduced the following theorem, which is an extension of a result of [31]. This representation will give us some intuition when devising a reverse engineering algorithm for PNCFs.

Theorem 4. *Let* $y_i = x_{\sigma(i)} + a_i + b_i$, $1 \leq i \leq d$ *and let*

$$f(x_1, \ldots, x_n) = y_1 \Diamond_1 (y_2 \Diamond_2 (\ldots (y_d \Diamond_d g(x_{\sigma(d+1)}, \ldots, x_{\sigma(n)})) \ldots)), \tag{4.3}$$

where

$$\Diamond_i = \begin{cases} \vee & if\ b_i = 1 \\ \wedge & if\ b_i = 0 \end{cases},$$

$a_i, b_i \in \{0, 1\}$ *for* $1 \leq i \leq d$, *and*

(i) None of the variables $x_{\sigma(d+1)}, \ldots, x_{\sigma(n)}$ *are canalyzing in* g, *or*

(ii) g *is a constant function.*

Then f *has canalyzing depth* d, *with canalyzing sequence* $x_{\sigma(1)}, \ldots, x_{\sigma(d)}$. *These variables have canalyzing values* $a_1, \ldots, a_d$ *and canalyzed values* $b_1, \ldots, b_d$. *Furthermore, any function of canalyzing depth* d *can be represented in this form.*

4.3.2 Algorithm Description

Using the NCF algorithm in [27] and the ideal of points method described in [40], we have devised an algorithm for the reverse engineering of networks using PNCFs. As input, we require a minimal wiring diagram, a discretized time series of network data, D, and the a list L of nested canalyzing inputs for each variable. This list may be found through prior knowledge of the system, or by restrictions in the data. If multiple possibilities exist, then the algorithm may be run multiple times using the different sets of canalyzing variables to obtain more possible solutions.

We begin by constructing a reduced wiring diagram. To do so, for each node in the wiring diagram, we remove all inputs not in L, the non-canalyzing inputs. Based on

this reduced wiring diagram, we partition the data points in our time series as follows. By the minimality of our wiring diagram, removing any inputs from our diagram, in this case the non-canalyzing variables, should result in inconsistencies in our data. If removing these variables does not result in inconsistencies, then our function only depends on the canalzying variables, thus violating our minimality assumption. Therefore, we partition D into points D_L that are consistent with the canalyzing variables and therefore may be described using the variables in L, and the points $D \setminus D_L$ that are inconsistent with the canalyzing variables, and therefore must be a function of the non-canalyzing variables.

Example 16. *Suppose in Table 4.2, our canalyzing variables are x_1 and x_2. Note that if we remove x_3 and x_4 from the wiring diagram, our network is inconsistent as the input $(x_1, x_2) = (0, 0)$ gives both 0 and 1 as output. Therefore, these data points must be explained by the non-canalzying variables x_3 and x_4, and are therefore included in the partition $D \setminus D_L$.*

Table 4.2: Sample time series data for Example 16

x_1	x_2	x_3	x_4	output
0	0	0	0	0
0	0	0	1	1

We then run the Hinkelmann and Jarrah algorithm on the partition D_L to obtain an ideal I of NCFs that interpolate the points that can be explained by the canalyzing variables. Finally, we run the Laubenbacher and Stigler algorithm on the partition $D \setminus D_L$ to obtain an ideal J that interpolates the remaining data points. Based on Theorem 4, a PNCF model that fits the data will be of the form $f \Diamond_d g$ where $f \in I$ and $g \in J$. Note that this list is not exhaustive, as we must specify the canalyzing variables as input. Alternative models may be found by varying these canalyzing variables as well as the function depth. A summary of our algorithm is as follows.

Algorithm 2. *Given minimal wiring diagram W, list L_i of canalyzing for each variable x_i, discretized time series data D,*

For each variable x_i:

1. *Eliminate inputs in* W *not included in* L_i *to obtain reduced wiring diagram* W_r
2. *Using* W_r*, partition* D *into* D_{L_i} *and* $D \setminus D_{L_i}$
3. *Run Hinkelmann-Jarrah algorithm on* D_{L_i} *and* W_r *and obtain all NCFs that fit this partition of the data,* I
4. *Run Laubenbacher-Stigler algorithm on* $D \setminus D_{L_i}$ *and obtain all functions that fit the remaining data,* J
5. *PNCF interpolating polynomial is of the form* $f_i = f \Diamond_d g$*, where* $f \in I$*,* $g \in J$

The bottlenecks for this algorithm are finding the minimal wiring diagram if it is unknown, and running the Hinkelmann-Jarrah and Laubenbacher-Stigler algorithms on the partitioned data. The stability properties of partially nested canalyzing functions and our ability to reverse engineer networks using this class of functions establishes and confirms their utility as biological network models.

Chapter 5

Conclusions and Discussion

5.1 Significance of Results

In this work, we have discussed two classes of Boolean functions with biologically significant properties that make them excellent candidates for the modeling of biological networks. In Chapter 2, we studied the nested canalyzing functions and their stability properties. We introduced two measures for analyzing network stability, the average height, which measures the average number of time steps until the system converges, and the average cycle length, which measures the length of the average limit cycle. We found that, on average, the nested canalyzing functions produced more stable networks with regard to these two measures than the general Boolean case in almost every case we considered. Our statistical analysis confirmed that our results are statistically significant.

In Chapter 3, we defined a new class of functions, the partially nested canalyzing functions, and analyzed their stability. The PNCFs are a necessary relaxation of the NCFs, as an NCF may not always fit the data, or may not reflect the experimentally observed behavior of the system under consideration. The PNCFs are characterized by their depth, which measures the extent to which the functions obtain the nested canalyzing structure. We introduced some properties of PNCFs and proved that the function depth is unique, regardless of the variable order. We were able to determine the activities of the variables

of partially nested canalyzing functions of depth at least d. We used this count to derive an expression for the expected sensitivities of PNCFs and found that increasing function depth decreases the expected sensitivity. Moreover, we found that the differences in expected sensitivities of functions of depth d and those of depth $d+1$ rapidly approach zero, so enforcing the rigid NCF structure is simply unnecessary. This observation is consistent with our intuition, as the first canalyzing variables are more influential on the function output. We applied PNCFs of varying depths to random Boolean networks and found similar patterns in the Derrida curves. As the depth increased, the functions rapidly approached the critical regime of stability, which is thought to be where many biological systems lie.

Finally, in Chapter 4, we introduced a method for the reverse engineering of biological networks using partially nested canalyzing functions. Given a minimal wiring diagram, a discretized time course of network data, and a list of nested canalyzing inputs for each variable, we designed an algorithm to produce PNCF models that interpolate the data.

5.2 Future Work

The application of nested and partially nested canalyzing functions to biological networks is a blossoming field, so there is still much work to be done in this area of research.

Our work in Chapter 2 motivates us to continue exploring other measures of stability for NCFs. Perhaps studying the number of fixed points or the number of connected components in the network could lead to interesting NCF properties. Alternative measures, such as the maximum cycle length or height could also yield significant results. We could explore how changing the function depth affects the network stability according to these measures. Ultimately, the development of concrete formulas or expressions for these measures, similar to the expected sensitivities formula, would be desirable.

In Chapter 3, we could explore how varying the distribution on the inputs in the wiring diagram affects the stability of our networks. In all of our Derrida plots, we used a constant number of inputs per function. In the future, we could impose a uniform, power

law, or exponential distribution on the inputs and observe the effects on stability. We could also apply these distributions to our methods in Chapter 2. In addition, the results in this chapter were based on two key assumptions. For the nested canalyzing portion of the function in $x_1, \ldots, x_d$, we assumed that all NCFs can occur with equal probability. In [35, 36], the authors select NCFs for their random networks according to a probability distribution. This distinction could perhaps explain the discrepancies between their stability results and those in Chapter 3. While they claim that networks constructed using NCFs are always stable, our Derrida plots indicate that these networks in fact lie in the critical regime. Second, we assumed that the remaining function in $x_{d+1}, \ldots, x_k$ was unbiased, i.e. that the probability p of a value of 1 in the truth table was 0.5. The dynamics of biased networks are discussed, for example, in [14]. In the future, studying the effects of varying the probability distributions on the selection of NCFs and the bias of the remaining function could lead to a more complete understanding of function dynamics.

In Chapter 4, the most important area of future study is reducing the algorithm inputs. We would like to build the network from the time series data without requiring prior knowledge of the canalyzing variables. Ideally, we would like to be able to generate all PNCFs of a given depth that fit the data without having to rerun the algorithm using each combination of canalyzing variables.

There are many extensions of this work that are applicable to all of the ideas that we have discussed. For instance, in [12, 11], the authors provide a definition of nested canalyzing functions over general fields. We could extend this definition to include partially nested canalyzing functions and explore the stability properties of NCFs and PNCFs in this broader framework. We could study the ideal of NCFs over a general field and develop reverse engineering algorithms for this case. Another interesting direction of study is that of binary decision diagrams. While the connection between unate cascade functions and nested canalyzing functions has been established, no implications regarding this result have been confirmed. For instance, does average path length on a binary decision diagram imply function stability? How does relaxing the nested canalyzing condition (i.e. to PNCFs)

affect the binary decision diagram? Are there alternative relaxations of NCFs that are more meaningful in terms of their binary decision diagrams? Also, all of our results were based on deterministic Boolean networks with parallel updating schemes. Certainly these methods could be applied to probabilistic Boolean networks and/or sequential dynamical systems.

While we have provided a basis for the study of NCFs and PNCFs as Boolean network models, there are still many directions to explore. We hope that the results presented in this work are a source of motivation for continuing the application and study of these promising classes of functions.

Appendices

Appendix A Header Files for NCF Simulations

A.1 NCF Class

The header file (fncf.h) for nested canalyzing functions used the in simulations for Chapter 2 is as follows.

```
#ifndef FNCF_H
#define FNCF_H

#include<vector>
#include<iostream>
#include<cstdlib>

using namespace std;

class fncf{
 public:

  //Constructor; input: number of variables
  fncf(int n){
    numvars = n;
    //Fills in sigma
    for (int i=0; i<numvars; i++) sigma.push_back(i);
    while(1){
      //Creates random permutation sigma
      permute();
      /*Fills in a and b vectors, which store canalyzing
        inputs and outputs*/
      for (int i=0; i<numvars; i++){
```

```
      a.push_back(rand()%2);
      b.push_back(rand()%2);
    }
    /*Check function eliminates bias in function generation;
      if check fails, we start over*/
    if (check()) return;
    else{
      a.resize(0);
      b.resize(0);
    }
  }
}

/*Verifies that sigma(n-1)<sigma(n) when b(n-1)=b(n) to
  eliminate bias in function generation*/
int check(){
  int hold = numvars-2;
  if (sigma[numvars-1] < sigma[numvars-2]) return 0;
  while(hold >= 1){
    if ((b[hold]==b[hold-1]) && (sigma[hold]<sigma[hold-1])){
      return 0;
    }
    hold--;
  }
  return 1;
}

//Generates a random permutation of the variables
```

```
void permute(){
  for (int j=numvars-1; j>=0; j--){
    swap(sigma[j], sigma[rand()%(j+1)]);
  }
}

/*Evaluates the function; input: integer representation of
  point at which function is to be evaluated*/
int ncfeval(int x){
  int ans, t, index = numvars-1;
  ans = (x >> sigma[index]) & 1;
  ans = (ans + a[index] + b[index]) & 1;
  index--;
  while (index >= 0){
    t = (x >> sigma[index]) & 1;
    t = (t + a[index] + b[index]) & 1;
    if (b[index]) ans |= t;
    else ans &= t;
    index--;
  }
  return ans;
}

/*Prints the function; takes integer input as index for
  state transition function in an FDS*/
void printfn(int iter){
  int index = 0;
  cout << 'f' << iter << " = (";
```

```
    while (index < numvars){
      if ((a[index]+b[index])%2) {
        cout << "(~x" << sigma[index]+1 <<  ')';
      }
      else cout << 'x'<< sigma[index]+1;
      if (index < numvars-1){
        if (b[index]) cout << "+(";
        else cout << "*(";
      }
      index++;
    }
    for (int i=0; i<numvars; i++) cout << ')';
    cout << endl;
  }
 private:
  /*Stores number of variables, permutation of the variables,
    canalyzing inputs and outputs*/
  int numvars;
  vector<int> sigma, a, b;
};

#endif
```

A.2 Boolean Function Class

The header file (fpoly.h) for Boolean functions used in the simulations for Chapter 2 is as follows. Functions are represented as polynomials over $\mathbb{F}_2$.

```
#ifndef FPOLY_H
```

```
#define FPOLY_H

#include<vector>
#include<iostream>
#include<cstdlib>

using namespace std;

class fpoly{
 public:
  //Constructor; input: number of variables
  fpoly(int n){
    numvars = n;
    int maxterms = (1<<n);
    for (int i=0; i<maxterms; i++){
      if (rand()%2) termvec.push_back(i);
    }
  }

  /*Evaluates the function; input: integer representation of
    point at which function is to be evaluated*/
  int polyeval(int x){
    int ans = 0;
    for (int i=0; i< termvec.size(); i++){
      if ((termvec[i]&x) == termvec[i]) ans++;
    }
    return (ans&1);
  }
```

```
/*Prints the function; takes integer input to print
  as a state transition function in an FDS*/
void printfn(int iter){
  int flag = 0;
  cout << 'f' << iter << " = ";
  for (int i=0; i<termvec.size(); i++){
    if (flag++>0) cout << " + ";
    //Constant term
    if (termvec[i] == 0){
      cout << "1";
      continue;
    }
    //Prints each term
    int hold = termvec[i], flag1 = 0;
    for(int j=0; j<numvars; j++){
      if (hold&1){
        if (flag1++>0) cout << '*';
        cout << "x" << j+1;
      }
      hold >>=1;
    }
  }
  cout << endl;
}

private:
/*Stores the number of variables and the terms with
```

```
    coefficient 1*/
  int numvars;
  vector<int> termvec;
};

#endif
```

Appendix B Header File for PNCF Simulations

The header file (ald.h) for partially nested canalyzing functions of depth at least d used to create the Derrida plots in Chapter 3 is as follows. Boolean functions are represented by their truth tables, which are stored as integer maps. While this construction is not as efficient as possible storage-wise, it allows for very efficient function evaluation, which is important in our simulations. Boolean networks are stored using a vector of ald's for the state transition functions and a matrix of integers for the wiring diagram. For each node in the network, its index in the ald vector gives its state transition function, and its corresponding row in the matrix stores its inputs in the wiring diagram.

```
#ifndef ALD_H
#define ALD_H

#include<map>
#include<vector>
#include<iostream>
#include<cstdlib>
#include<ctime>

using namespace std;

typedef map<int,int> intmap;
typedef intmap::iterator mapitr;

class ald{
 public:
  //Constructor; input: number of variables and depth
  ald(int n, int d){
```

```
numvars = n;
depth = d;
int expn = (1<<numvars);
if (depth > n || depth < 0 || depth == n-1){
  cerr << "Improper value of d" << endl;
  return;
}
else if (depth == 0){
  for (int i=0; i<expn; i++){
    table[i] = rand()%2;
  }
}
else {
  while(1){
    //Pick d nested canalyzing variables
    for (int i=0; i<numvars; i++) sigma.push_back(i);
    permute();
    for (int i=numvars-1; i>depth-1; i--){
      //notsigma stores noncanalyzing variables
      notsigma.push_back(sigma.back());
      sigma.pop_back();
    }
    //Select canalyzing input/output values
    for (int i=0; i<depth; i++){
      a.push_back(rand()%2);
      b.push_back(rand()%2);
    }
    /*Check function eliminates bias; if check fails, we
```

```
        start over*/
      if (check()) break;
      else{
        a.clear();
        b.clear();
        sigma.clear();
        notsigma.clear();
      }
    }
    //Creates "empty" truth table
    for (int j=0; j<(1<<numvars); j++) table[j] = -1;
    //Fills in truth table
    filltable();
  }
}

/*Verifies that sigma(n-1)<sigma(n) when b(n-1)=b(n) to
  eliminate bias in function generation*/
int check(){
  if (depth < 3) return 1;
  int hold = depth-2;
  if (sigma[depth-1] < sigma[depth-2]) return 0;
  while(hold >= 1){
    if ((b[hold]==b[hold-1]) && (sigma[hold]<sigma[hold-1])){
      return 0;
    }
    hold--;
  }
```

```
  return 1;
}

//Permutes the variables
void permute(){
  for (int j=numvars-1; j>=0; j--){
    swap(sigma[j], sigma[rand()%(j+1)]);
  }
}

//Fills in truth table output values
void filltable() {
  int tsize = table.size();
  //Fills in canalyzing truth table output values
  for (int i=0; i<depth; i++){
    mapitr ttitr;
    for (ttitr = table.begin(); ttitr != table.end(); ttitr++){
      if (((*ttitr).second == -1) &&
      (((ttitr->first)>>sigma[i])&1) == a[i]){
        ttitr->second = b[i];
      }
    }
  }
  //Fills in remaining truth table values
  for (int i=0; i<tsize; i++){
    if (table[i] == -1) table[i] = rand()%2;
  }
}
```

```
  //Prints truth table
  void printald(){
    mapitr pitr;
    for (pitr = table.begin(); pitr != table.end(); pitr++){
      /*for (int i=numvars-1; i>=0; i--){
        cout << (((pitr->first)>>i)&1);
        }*/
      cout << pitr->second << ' ';
    }
    cout << endl;
  }
 private:
  //Stores number of variables, function depth
  int numvars, depth;
  //Stores the truth table
  intmap table;
  /*Stores permutation of canalyzing variables, canalyzing
    inputs and outputs, and noncanalyzing variables*/
  vector<int> sigma, a, b, notsigma;
};

#endif
```

Appendix C Code for Creating Derrida Plots

The C++ code for creating the Derrida Plots in Chapter 3 is included. Inputs are k, or number of inputs per function, d, the depth of each function, and t, the number of trials. This code creates 25 random Boolean networks, each with 100 nodes and k inputs per node, and uses functions of depth at least d. For each network, it selects t points. For each point, it finds a random permutation of the point, stores the normalized Hamming distance $p(t)$ between the points, evaluates the points, and stores the normalized Hamming distance $p(t+1)$ between the evaluated points. It then computes the mean and standard deviation for each value of $p(t)$ over all points and networks. We created alternative versions of this code using different methods described in the literature and obtained analogous results.

```
#include<cstdlib>
#include<map>
#include<iostream>
#include<vector>
#include<cstdio>
#include<cmath>
#include<fstream>
#include"ald.h"

using namespace std;

typedef map<int,int> intmap;
typedef vector<vector<int>> matrix;
typedef vector<int> intvec;
typedef vector<vector<float>> flmatrix;
typedef vector<intmap> mapvec;
typedef map<int,int>::iterator mapitr;
```

```
typedef vector<float> flvec;

void fillvars(matrix &, int, int);
void permute(intvec &, int, int stop = 0);
void fillfuns(mapvec &, int, int, int);
void fillcounts(flmatrix &, flmatrix &, matrix,
                mapvec, int, int, int);
void print(flmatrix &, flvec &, int, ofstream &);
int hamming(intvec, intvec, int);
void points(intvec &, intvec &, int);
intvec evaluate(matrix &, mapvec &, intvec &, int, int);

int main(int argc, char **argv){
  srand(time(0));
  if (argc < 3){
    cerr << "Usage: k d t" << endl;
    return -1;
  }
  /*k = number of inputs per function,
    d = depth of each function,
    n = number of variables in RBN,
    t = number of trials,
    net = number of networks to generate
  */
  int k, d, n=100, t, net=25;
  sscanf(argv[1], "%d", &k);
  sscanf(argv[2], "%d", &d);
  sscanf(argv[3], "%d", &t);
```

```
char buffer[256];
sprintf(buffer, "kauffman_%d.txt", d);
ofstream outfile(buffer);
outfile << "x_=_[";
for(int i=0; i<=n; i++){
  outfile << i*(1.0)/n << '_';
}
outfile << "];" << endl;

//Input variables for each function
outfile << "Derrida_curve_for_" << n << "_variables;_";
outfile << k << "_inputs_per_variable;_";
outfile << "functions_of_depth_" << d << endl;
flmatrix counts(n+1);
flmatrix values(n+1);
flvec stdev(n+1);
//Index is p(t)*n (non-normalized Hamming distance)
//counts[i][0] is sum of p(t+1)*n for p(t)*n = i
//counts[i][1] is number of trials for which p(t)*n = i
for (int i=0; i<=n; i++){
  counts[i].resize(2);
  counts[i][0] = counts[i][1] = 0;
  values[i].resize(0);
  stdev[i] = 0;
}
//Creates and collects data on net networks
for (int g=0; g<net; g++){
  //Stores the input variables for each node
```

```
    matrix invars(n);
    //Randomly fills the input variables
    fillvars(invars, n, k);
    //Stores state transition functions for each node
    mapvec functions(n);
    /*Randomly fills the state transition functions using
      ald class*/
    fillfuns(functions, n, k, d);
    //Collects data on the network
    fillcounts(counts, values, invars, functions, n, k, t);
  }
  //Computes sample means for each value of p(t)
  for (int i=1; i<=n; i++){
      counts[i][0] /= counts[i][1];
  }
  //Computes sample standard deviations for each value of p(t)
  for (int l=0; l<=n; l++){
    float sigma = 0;
    for (int j=0; j<values[l].size(); j++){
      sigma += pow(values[l][j] - counts[l][0], 2);
    }
    stdev[l] = sqrt(sigma/(counts[l][1]-1));
  }
  print(counts, stdev, n, outfile);

  return 0;
}
```

```
/*Randomly assigns the input variables for each node;
  each node has k inputs*/
void fillvars(matrix &invars, int n, int k){
  static intvec temp;
  if (temp.empty()){
    for(int i=0; i<n; i++) temp.push_back(i);
    permute(temp, n);
  }
  //Each node gets k inputs
  for (int j=0; j<n; j++){
    permute(temp, n, n-k);
    for (int l=0; l<k; l++) invars[j].push_back(temp[n-l-1]);
  }
}

//Permutes a vector of integers
void permute(intvec &temp, int n, int stop){
  for (int j=n-1; j>=stop; j--){
    swap(temp[j], temp[rand()%(j+1)]);
  }
}

//Randomly assigns state transition function to each node
void fillfuns(mapvec &functions, int n, int k, int d){
  for (int i=0; i<n; i++){
    ald temp(k, d);
    functions[i] = (temp.table);
  }
```

```
}

//Picks a random point x1 and perturbs it to obtain x2
void points(intvec &x1, intvec &x2, int n){
  x1.clear();
  x2.clear();
  for (int i=0; i<n; i++){
    //Selects a random point x1
    x1.push_back(rand()%2);
    x2.push_back(x1[i]);
  }
  //Selects number of bits of x1 to flip
  int bflips = (rand()%n)+1;
  //Selects which bits of x1 to flip
  intvec temp;
  for (int m=0; m<n; m++) temp.push_back(m);
  permute(temp, n);
  /*Flips the selected bits and stores the perturbed
    point as x2*/
  for (int j=0; j<bflips; j++){
    x2[temp[j]] = (x1[temp[j]]+1)%2;
  }
}

void fillcounts(flmatrix &counts, flmatrix &values, matrix invars,
                mapvec functions, int n, int k, int t){
  //Two points used to calculate p(t) and p(t+1)
  intvec x1, x2;
```

```
  for (int trial = 0; trial<t; trial++){
    //Randomly selects point x1 and perturbation x2
    points(x1, x2, n);
    //Computes and stores Hamming distance between x1 and x2
    float rhot = hamming(x1, x2, n);
    counts[rhot][1] += 1;
    //Evaluate x1, x2 using state transition functions
    intvec x1e = evaluate(invars, functions, x1, n, k);
    intvec x2e = evaluate(invars, functions, x2, n, k);
    //Computes Hamming distance between evaluated points
    float rhot1 = hamming(x1e, x2e, n);
    //Stores normalized Hamming distance
    counts[rhot][0] += rhot1/n;
    values[rhot].push_back(rhot1/n);
  }
}


//Computes Hammning distance between points vec1 and vec2
int hamming(intvec vec1, intvec vec2, int n){
  int ans = 0;
  for (int i=0; i<n; i++){
    ans += (vec1[i] + vec2[i])%2;
  }
  return ans;
}

//Prints mean normalized Hamming distances and standard deviations
```

```
void print(flmatrix &counts, flvec &stdev, int n,
           ofstream &outfile){
  outfile << "y = [";
  for (int j=0; j<=n; j++){
    //Normalized Hamming distances
    outfile << counts[j][0] << ' ';
  }
  outfile << "];" << endl;
  outfile << "stdev = [";
  for (int j=0; j<=n; j++){
    outfile << stdev[j] << ' ';
  }
  outfile << "];" << endl;
}

//Evaluates Boolean network at a point p
intvec evaluate(matrix &invars, mapvec &functions,
                intvec &p, int n, int k){
  intvec ans(n);
  for (int l=0; l<n; l++){
    int pnt = 0;
    for (int r=0; r<k; r++){
      pnt |= (p[invars[l][r]])<<r;
    }
    ans[l] = functions[l][pnt];
  }
  return ans;
}
```

Bibliography

[1] J. Abbott, A. Bigatti, M. Kreuzer, and L. Robbiano. Computing ideals of points. *J. Symb. Comput.*, 30(4):341–356, 2000.

[2] R. Albert and H. Othmer. The topology of the regulatory interactions predicts the expression pattern of the segment polarity genes in *Drosophila melanogaster*. *J. Theor. Bio.*, 223:1–18, 2003.

[3] M. Aldana. Boolean dynamics of networks with scale-free topology. *Physica D*, 185:45–66, 2003.

[4] E. Balleza, E. Alvarez-Buylla, A. Chaos, S.A. Kauffman, I. Shmulevich, and M. Aldana. Critical dynamics in genetic regulatory networks: Examples from four kingdoms. *PLoS ONE*, 3(6):e2456, 2008.

[5] D. Bollman, O. Colón-Reyes, V. Ocasio, and E. Orozco. A control theory for Boolean monomial dynamical systems. *Discrete Event Dyn. S.*, 20:19–35, 2010.

[6] J.T. Butler, T. Sasao, and M. Matsuura. Average path length of binary decision diagrams. *IEEE Trans. Comput.*, 54(9):1041–1053, 2005.

[7] F. Celada and P. Seiden. A computer model of cellular interactions in the immune system. *Immunol. Today*, 13:56–62, 1992.

[8] O. Colón-Reyes, R. Laubenbacher, and B. Pareigis. Boolean monomial dynamical systems. *Ann. Comb.*, 8:425–439, 2005.

[9] D.A. Cox, J. Little, and D. O'Shea. *Ideals, Varieties, and Algorithms: An Introduction to Computational Algebraic Geometry and Commutative Algebra, 3/e (Undergraduate Texts in Mathematics)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2007.

[10] D.A. Cox, J. Little, and H. Schenck. *Toric Varieties*. Preprint available at: http://www.cs.amherst.edu/ dac/toric/toric.pdf, 2010.

[11] R. Laubenbacher D. Murrugarra. The number of multistate nested canalyzing functions. In preparation.

[12] R. Laubenbacher D. Murrugarra. Regulatory motifs in molecular interaction networks. *arXiv:1102.3739v2*, 2011.

[13] E. Davidson and M. Levin. Gene regulatory networks. *Proc. Natl. Acad. Sci.*, 102:4935, 2005.

[14] B. Derrida and Y. Pomeau. Random networks of automata: a simple annealed approximation. *Europhys. Lett.*, 1:45–49, 1986.

[15] E.S. Dimitrova, A.S. Jarrah, R.C. Laubenbacher, and B. Stigler. A Gröbner fan method for biochemical network modeling. In *ISSAC'07*, pages 122–126, 2007.

[16] E.S. Dimitrova, P. Vera-Licona, .J McGee, and R. Laubenbacher. Discretization of time series data. *J. Comput. Biol.*, 17(6), 2010.

[17] E.S. Dimitrova and O. Yordanov. In preparation.

[18] B. Drossel. *Random Boolean Networks*, chapter 3, pages 69–110. Wiley-VCH Verlag GmbH & Co., Weinheim, Germany, 2009.

[19] D. Eisenbud and B. Sturmfels. Binomial ideals. *Duke Math. J.*, 84(1):1–45, 1996.

[20] J. Faugère. A new efficient algorithm for computing gröbner bases (f4). *J. Pure Appl. Algebra*, 139:61–88, 1999.

[21] A. Gambin, S. Lasota, and M. Rutkowski. Analyzing stationary states of gene regulatory network using Petri nets. *Silico Biology*, 6:93–109, 2006.

[22] S. Gao, Y. Guan, and F. Volny IV. A new incremental algorithm for computing groebner bases. In *Proceedings of the 2010 International Symposium on Symbolic and Algebraic Computation*, ISSAC '10, pages 13–19, New York, NY, USA, 2010. ACM.

[23] I. Gat-Viks and R. Shamir. Chain functions and scoring functions in genetic networks. *Bioinformatics*, 19:108–117, 2003.

[24] R. Gebauer and H. M. Möller. On an installation of buchberger's algorithm. *J. of Symb. Comput.*, 6:275–286, 1988.

[25] A. Hartemink. *Principled Computational Methods for the Validation and Discovery of Genetic Regulatory Networks*. PhD thesis, Massachusetts Institute of Technology, 2001.

[26] F. Hinkelmann. Personal communication, 2011.

[27] F. Hinkelmann and A.S. Jarrah. Inferring biologically relevant models: Nested canalyzing functions. *arXiv:1011.6064v1*, 2010.

[28] A.S. Jarrah and R. Laubenbacher. Discrete models of biochemical networks: The toric variety of nested canalyzing functions. In H. Anai, K. Horimoto, and T. Kutsia, editors, *Algebraic Biology*, volume 4545 of *Lecture Notes in Computer Science*, pages 15–22. Springer Berlin / Heidelberg, 2007.

[29] A.S. Jarrah, R. Laubenbacher, B. Stigler, and M. Stillman. Reverse-engineering of polynomial dynamical systems. *Adv. Appl. Math.*, 39:477–489, 2007.

[30] A.S. Jarrah, R. Laubenbacher, and A. Veliz-Cuba. The dynamics of conjunctive and disjunctive Boolean network models. *B. Math. Biol.*, 72:1425–1447, 2010.

[31] A.S. Jarrah, B. Raposa, and R. Laubenbacher. Nested canalyzing, unate cascade, and polynomial functions. *Physica D*, 233:167–174, 2007.

[32] F. Karlssona and M. Hörnquist. Order or chaos in Boolean gene networks depends on the mean fraction of canalyzing functions. *Physica A*, 384:747–757, 2007.

[33] S.A. Kauffman. Metabolic stability and epigenesis in randomly constructed genetic nets. *J. Theor. Biol.*, 22(3):437–467, 1969.

[34] S.A. Kauffman. *The Origins of Order: Self-Organization and Selection in Evolution.* Oxford Universiy Press, 1993.

[35] S.A. Kauffman, C. Peterson, B. Samuelsson, and C. Troein. Random Boolean network models and the yeast transcriptional network. *Proc. Natl. Acad. Sci.*, 100(25):14796–9, 2003.

[36] S.A. Kauffman, C. Peterson, B. Samuelsson, and C. Troein. Genetic networks with canalyzing Boolean rules are always stable. *Proc. Natl. Acad. Sci.*, 101(49):17102–17107, 2004.

[37] B. Krupa. On the number of experiments required to find the causal structure of complex systems. *J. Theor. Biol.*, 219:257–267, 2002.

[38] R. Laubenbacher, A. Jarrah, H. Vastani, and B. Stigler. Discrete visualizer of dynamics.
http://dvd.vbi.vt.edu/.

[39] R. Laubenbacher and B. Pareigis. Finite dynamical systems. Technical report, Department of Mathematical Sciences, New Mexico State University, Las Cruces, 2000.

[40] R. Laubenbacher and B. Stigler. A computational algebra approach to the reverse engineering of gene regulatory networks. *J. Theor. Biol.*, 229(4):523–537, 2004.

[41] T.I. Lee, N.J. Rinaldi, F. Robert, D.T. Odom, Z. Bar-Joseph, G.K. Gerber, N.M. Hannett, C.R. Harbison, C.M. Thompson, I. Simon, J. Zeitlinger, E.G Jennings, H.L. Murray, D.B. Gordon, B. Ren, J.J. Wyrick, J. Tagne, T.L. Volkert, E. Fraenkel, D.K. Gifford, and R.A. Young. Transcriptional regulatory networks in Saccharomyces cerevisiae. *Science*, 298:799–804, 2002.

[42] F. Li, T. Long, Y. Lu, Q. Ouyang, and C. Tang. The yeast cell-cycle network is robustly designed. *Proc. Natl. Acad. Sci.*, 11:4781–4786, 2004.

[43] J.F. Lynch. On the threshold of chaos in random Boolean cellular automata. *Random Struct. Algorithms*, 6(2–3):239–260, 1995.

[44] D. Maclagan, R.R. Thomas, T. Puthenpurackel, A.V. Jayanthan, A. Khetan, L. Gold, and S. Faridi. Lecture notes for the graduate school at the international conference on commutative algebra & combinatorics h.r.i., allahabad, december 8-13, 2003, 2003.

[45] E.W. Mayr and A.R. Meyer. The complexity of the word problems for commutative semigroups and polynomial ideals. *Advances in Mathematics*, 46:305–329, 1982.

[46] H.S. Mortveit and C.S. Reidys. *An Introduction to Sequential Dynamical Systems*. Springer Science+Business Media LLC, 2008.

[47] K. Nagel and P. Wagner. A cellular automaton model for freeway traffic. *J. Phys. I*, 2:2221–2229, 1992.

[48] S. Nikolajewa, M. Friedel, and T. Wilhelm. Boolean networks with biologically relevant rules show ordered behavior. *Biosystems*, 2006.

[49] M. Nykter, N.D. Price, M. Aldana, S.A. Ramsey, S.A. Kauffman, L.E. Hood, O. Yli-Harja, and I. Shmulevich. Gene expression dynamics in the macrophage exhibit criticality. *Proc. Natl. Acad. Sci.*, 105:1897–1900, 2008.

[50] M. Nykter, N.D. Price, A. Larjo, T. Aho, S.A. Kauffman, O. Yli-Harja, and I. Shmulevich. Critical networks exhibit maximal information diversity in structure-dynamics relationships. *Phys. Rev. Lett.*, 100:058702, 2008.

[51] T.P. Peixoto. The phase diagram of random Boolean networks with nested canalizing functions. *Eur. Phys. J. B*, 78:187–192, 2010.

[52] L. Raeymaekers. Dynamics of Boolean networks controlled by biologically meaningful functions. *J. Theor. Biol.*, 218:331–341, 2002.

[53] J. Saez-Rodriguez, L. Simeoni, J. Lindquist, R. Hemenway, U. Bommhardt, B. Arndt, U. Haus, R. Weismantel, E. Gilles, S. Klamt, and B. Schraven. A logical model provides insights into T cell receptor signaling. *PLoS Comput. Biol.*, 3:e163, 2007.

[54] I. Shmulevich, E.R. Dougherty, S. Kim, and W. Zhang. Probabilistic Boolean networks: A rule-based uncertainty model for gene regulatory networks. *Bioinformatics*, 18(2):261–274, 2002.

[55] I. Shmulevich, E.R. Dougherty, and W. Zhang. From Boolean to probabilistic Boolean networks as models of genetic regulatory networks. *P. IEEE*, 90:1778–1792, 2002.

[56] I. Shmulevich and S.A. Kauffman. Activities and sensitivities in Boolean network models. *Phys. Rev. Lett.*, 93(4):048701, 2004.

[57] I. Shmulevich, S.A. Kauffman, and M. Aldana. Eukaryotic cells are dynamically ordered or critical but not chaotic. *Proc. Natl. Acad. Sci.*, 102:13439–13444, 2005.

[58] J. von zur Gathen and J. Gerhard. *Modern Computer Algebra*. Cambridge University Press, New York, NY, USA, 2 edition, 2003.

[59] C.H. Waddington. Canalisation of development and the inheritance of acquired characters. *Nature*, 150:563–564, 1942.