RUDZINSKI, JAMES, Ph.D. The Bunk Bed Conjecture and the Skolem Problem. (2021)
Directed by Dr. Clifford Smyth. 79 pp.

We present our results on two open problems, the Bunk Bed Conjecture and the Skolem Problem. The 35-year-old Bunk Bed Conjecture is a natural conjecture on connection events in a randomly disrupted network. We reduced this problem to a counting problem on graphs. As part of that research, we developed a new algorithm for calculating the inverse images of a monotone Boolean function. This algorithm greatly improves the space complexity of the existing Hansel's algorithm for this problem. It is also parallelizable whereas Hansel's algorithm is not. The 85-year-old Skolem Problem is to prove or disprove the existence of a decision procedure to determine whether a rational linear recurrence ever has a zero. We give a partial decision procedure for the more general problem of deciding whether a rational linear recurrence is non-negative. We give wide-ranging conditions under which our procedure is guaranteed to terminate.

THE BUNK BED CONJECTURE AND THE SKOLEM PROBLEM

by

James Rudzinski

A Dissertation Submitted to
the Faculty of The Graduate School at
The University of North Carolina at Greensboro
in Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy

Greensboro
2021

Approved by

_____
Committee Chair

This dissertation is dedicated to my wonderful wife Sandi, whose support has been immeasurable, and to our daughters Claire and Maycie. Thank you all for all of the love, encouragement, and patience you have shown me throughout this endeavor.

APPROVAL PAGE

This dissertation written by James Rudzinski has been approved by the following committee of the Faculty of The Graduate School at The University of North Carolina at Greensboro.

Committee Chair _____
Clifford Smyth

Committee Members _____
Gregory Bell

_____
Igor Erovenko

_____
Sebastian Pauli

_____
Date of Acceptance by Committee

_____
Date of Final Oral Examination

ACKNOWLEDGMENTS

James Rudzinski was partially supported by funds from Clifford Smyth's grants: NSA MSP Grant H98230-13-1-0222 and a Simons Collaboration Grant 360486.

I would like to thank Dr. Clifford Smyth for his tremendous support over the years. It is truly hard to express how grateful I am for everything he has done. At different times he has been encouraging, supportive, patient, helpful, compassionate, and caring. He has helped me not only to learn to be a better mathematician but a better person as well. It has been a long journey with many ups and downs, but we finally managed to finish this chapter. I look forward to the chapters ahead.

I would also like to thank the committee members. Dr. Sebastian Pauli was particularly helpful with our discussions about algorithms and complexity analysis. Dr. Sebastian Pauli, Dr. Gregory Bell, and Dr. Igor Erovenko have all been teachers to me at one time, and I know that I have gained something important from each one of those experiences. They have helped shape me into the mathematician I am now, and I will continue to try to live up to their examples.

Finally, I would like to thank the UNCG Department of Mathematics and Statistics for their support throughout the years.

Table of Contents

List of Figures

Chapter 1: Introduction

We present our results on two open problems in discrete mathematics, the Bunk Bed Conjecture and the Skolem Problem.

The 35-year-old Bunk Bed Conjecture states that if two isomorphic networks, subject to identical but independent probabilistic connection disruption processes, have some pairs of sites identified, then the probability that two sites x and y in one network remain connected is at least as large as the probability that sites x and z remain connected where z is the isomorphic copy of y in the second network. We prove that it is sufficient for the connection disruption probabilities to be all taken to be $1/2$ in order to prove this conjecture in full generality. This reduces the problem to a counting problem over all possible sub-networks of the joint network.

To facilitate computer testing of this counting version of the Bunk Bed Conjecture, we found a novel algorithm for computing the ideal $f^{-1}(0)$ of a monotone Boolean function, $f : B_n \to \{0, 1\}$ where $B_n = \{0, 1\}^n$ is the Boolean lattice. Our algorithm has space complexity that is polynomial in n as opposed to the exponential space complexity of the classical Hansel's algorithm for this problem. In addition, our algorithm is easily parallelizable, whereas Hansel's algorithm is not. This algorithm is also of independent interest in the problem of counting the number of simplices in an algorithmically defined abstract simplicial complex. We arrived at our results by the development of novel algorithms for scanning through all the elements of the Boolean lattice, following its so-called Christmas Tree Decomposition into symmetric chains.

The 85-year-old Skolem Problem is to find a decision procedure to determine whether a rational linear recurrence ever has a zero term or to prove that this is undecidable, i.e. that no such decision procedure can exist. This problem can be reduced to the more general problem of deciding whether a rational linear recurrence is non-negative or not. We give a novel partial decision procedure for the non-negativity problem. While the procedure does not always terminate in all the cases where the recurrence is non-negative, we give wide-ranging conditions under which it is guaranteed to terminate.

We present our results on the Bunk Bed Conjecture in Chapter 2, our results on computing the ideals of monotone Boolean functions in Chapter 3, and our results

on the Skolem problem in Chapter 4. We conclude with some directions for future research in Chapter 5.

Chapter 2: The Bunk Bed Conjecture

2.1 Introduction

A *bunk bed graph* consists of two isomorphic graphs, called the *upper and lower bunks*, and some additional edges, called *posts*; each post connects a vertex in the upper bunk with the corresponding isomorphic vertex in the lower bunk. We assign a probability to each edge with each edge in the upper bunk assigned the same probability as the corresponding isomorphic edge in the lower bunk. The probabilities on the posts are arbitrary. We then form a random spanning subgraph of the bunk bed graph by deleting each edge independently with its prescribed probability.

The Bunk Bed Conjecture states that in the random subgraph the probability that a vertex x in the upper bunk is connected to some vertex y in the upper bunk is greater than or equal to the probability that x is connected to z, the isomorphic copy of y in the lower bunk. We show that for each $p \in (0, 1)$ the special case of the Bunk Bed Conjecture in which all edge probabilities are p is equivalent to the full conjecture.

We now give the formal definitions of these concepts and the statements of these theorems.

Let G be a graph and let G_0 and G_1 be two disjoint isomorphic copies of G. Fix graph isomorphisms from G to G_0 and from G to G_1. If $x \in V(G)$ and $e \in E(G)$, then for $i \in \{0, 1\}$, let x_i and e_i be the isomorphic copies of x and e in G_i with respect to those isomorphisms. Let T be a subset of $V(G)$. The *bunk bed graph* $B = BB(G, T)$ is the graph consisting of G_0 and G_1 together with an edge $\{x_0, x_1\}$ for each $x \in T$. G_0 and G_1 are called the *upper* and *lower bunks* of B. We also define $E_T(B) = \{\{x_0, x_1\} : x \in T\}$ to be the set of *posts* of B.

The terms bunk and post are used because the following "3-dimensional drawing" of B is meant to evoke a bunk bed. Identical drawings of G_0 and G_1 are placed in two horizontal planes with each vertex x_0 of G_0 positioned directly above the corresponding vertex x_1 of G_1. This drawing of B looks a little like the drawing of a bunk bed with the upper bunk G_0 held above the lower bunk G_1 by the posts in $E_T(B)$. See Figures 2.1 and 2.2 below.

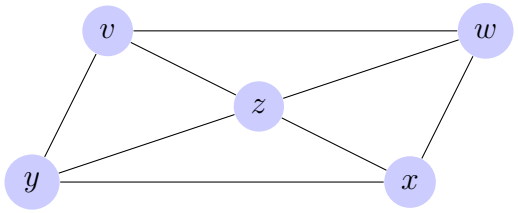

Figure 2.1. A graph G.

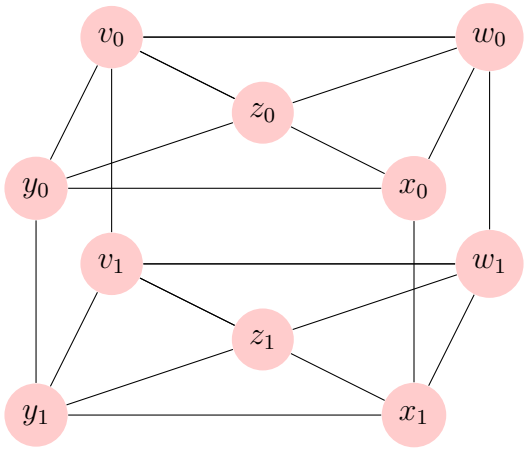

Figure 2.2. The graph $B = BB(G, \{v, w, x, y\})$.

Note that the vertices of the upper bunk of B in Figure 2.2 are v_0, w_0, x_0, y_0, z_0 and those of the lower bunk are v_1, w_1, x_1, y_1, z_1.

An *edge-probability function for a graph* G is a function $\mathbf{p} : E(G) \to [0, 1]$. If $B = BB(G, T)$ is a bunk bed graph, we say that $\mathbf{p} : E(B) \to [0, 1]$ is *symmetric* if for all edges $e \in E(G)$, $\mathbf{p}(e_0) = \mathbf{p}(e_1)$. Note that even if $\mathbf{p}$ is symmetric, this implies no restriction on $\mathbf{p}(e)$ if $e \in E_T(B)$.

We define $G_{\mathbf{p}}$ to be the random spanning subgraph of G that is obtained from G by the following edge percolation process: for each edge $e \in E(G)$, independently delete e with probability $1 - \mathbf{p}(e)$. Thus $G_{\mathbf{p}}$ takes on values in

$$\text{span}(G) = \{H : H \text{ is a spanning subgraph of } G\}.$$

If $p \in [0, 1]$ we write G_p to denote $G_{\mathbf{p}}$ where $\mathbf{p}(e) = p$ for all $e \in E(G)$.

If $x, y \in V(G)$, we write "$x \leftrightarrow y$ in $G_{\mathbf{p}}$" for the event that $G_{\mathbf{p}}$ contains a path from x to y.

We may now state the following conjecture.

Conjecture 2.1 (The Bunk Bed Conjecture)**.**

Let G be a graph, $T \subseteq V(G)$, and $B = BB(G,T)$. Let $\mathbf{p} : E(B) \to [0,1]$ be symmetric. Then, for all $x, y \in V(G)$, we have

$$P(x_0 \leftrightarrow y_0 \text{ in } B_{\mathbf{p}}) \geq P(x_0 \leftrightarrow y_1 \text{ in } B_{\mathbf{p}}).$$

This conjecture and several special cases have been part of probability folklore since 1985 or earlier; see Section 1, note 5 of [vdBK01] or Conjecture 2.1 of [Lin11]. Although Conjecture 2.1 is intuitively plausible and some special cases have been proved (it is true if G is outerplanar [Lin11]), no complete proof has been found.

We now state two special cases of Conjecture 2.1, each obtained by restricting to constant valued $\mathbf{p}$.

Conjecture 2.2 (The uniform version of the Bunk Bed Conjecture)**.**

Fix $p \in (0,1)$. Let G be a graph, $T \subseteq V(G)$, and $B = BB(G,T)$. Then for all $x, y \in V(G)$ we have

$$P(x_0 \leftrightarrow y_0 \text{ in } B_p) \geq P(x_0 \leftrightarrow y_1 \text{ in } B_p).$$

If we take $p = 1/2$ in Conjecture 2.2 we get:

Conjecture 2.3 (The counting version of the Bunk Bed Conjecture)**.**

Let G be a graph, $T \subseteq V(G)$, and $B = BB(G,T)$. Then for all $x, y \in V(G)$ the number of spanning subgraphs of B in which x_0 and y_0 are connected is at least the number of spanning subgraphs in which x_0 and y_1 are connected.

If G is a graph, an orientation of G is an assignment of a direction to each edge of G. Let $\vec{G}$ be an orientation of G chosen uniformly at random from all of the possible orientations of G. If $x, y \in V(G)$ we may then speak of events such as "$x \to y$ in $\vec{G}$", the event that there is a directed path from x to y in $\vec{G}$. The following theorem of [McD80] and [Kar90] is of interest.

Theorem 2.4. *Let G be a graph and $x, y \in V(G)$. Then for all $x, y \in V(G)$ we have*

$$P(x \leftrightarrow y \text{ in } G_{1/2}) = P(x \to y \text{ in } \vec{G}).$$

With this result in hand, the following conjecture is equivalent to Conjecture 2.3.

Conjecture 2.5 (The directed version of the Bunk Bed Conjecture)**.**

Let G be a graph, $T \subseteq V(G)$, and $B = BB(G,T)$. Then for all $x, y \in V(G)$ we have

$$P(x_0 \to y_0 \text{ in } \vec{B}) \geq P(x_0 \to y_1 \text{ in } \vec{B}).$$

The main results of this chapter are the following theorem and its immediate corollary.

Theorem 2.6. *If $p \in (0,1)$ is fixed and*

$$P(x_0 \leftrightarrow y_0 \text{ in } BB(G,T)_p) \geq P(x_0 \leftrightarrow y_1 \text{ in } BB(G,T)_p)$$

for all graphs G, all subsets $T \subseteq V(G)$, and all vertices $x, y \in V(G)$, then Conjecture 2.1 is true.

Corollary 2.7. *Conjectures 2.1, 2.2, 2.3, and 2.5 are all equivalent to one another.*

We prove Theorem 2.6 by simulation: we create a sequence of new bunk bed graphs B_n from B by replacing the edges of B with other graphs in such a way that

$$P(u \leftrightarrow v \text{ in } (B_n)_p) \to P(u \leftrightarrow v \text{ in } B_{\mathbf{p}}) \text{ as } n \to \infty,$$

where the convergence is uniform over all choices of vertices u, v that are original vertices of B.

We summarize this technique of simulation by edge-replacements in Lemmas 2.9 and 2.10 in Section 2.2. We then prove Theorem 2.6 in Section 2.3.

Notes:

(i) Conjecture 2.2 is an often-mentioned variant of Conjecture 2.1, see [Lin11] for example. The equivalence of the two apparently has never been noted in print. We believe the statement of Conjecture 2.5 is new. Linnusson uses Theorem 2.4 to formulate another conjecture related to Conjecture 2.8 below [Lin11].

(ii) In Conjecture 2.1, it is equivalent to assume that all posts in $BB(G,T)$ have probability 1. This is because the posts $\{x_0, x_1\}$ of $BB(G,T)$ may be replaced by internally vertex disjoint paths x_0, x_0', x_1', x_1 where x_0' and x_1' are added vertices, $\{x_0', x_1'\}$ is a new post of probability 1 and edges $\{x_0, x_0'\}$ and $\{x_1', x_1\}$ of the new upper and lower bunks respectively are each given probability $\sqrt{p}$. This may also be seen by applying the law of total probability, conditioning on the outcomes of the random variable $E(B_{\mathbf{p}}) \cap E_T(B)$. In Conjectures 2.2 and 2.3 it is likewise equivalent to assume that posts are always present.

(iii) One may also define $BB_i(G,T)$, the bunk bed graph consisting of G_0 and G_1 with x_0 and x_1 identified for each $x \in T$ instead of connected by a post. Note that $BB_i(G,T)$ may be a multi-graph; an edge $e = \{x, y\} \subseteq T$ will become two distinct edges e_0 and e_1 in $BB_i(G,T)$. Since posts may assumed to be always present, the statements of Conjectures 2.1, 2.2, 2.3, and 2.5 with $BB(G,T)$ replaced by $BB_i(G,T)$ are also all equivalent to each other and Conjecture 2.1. In fact, Conjecture 2.1 is stated in this form in [vdBK01].

(iv) The following special case of Conjecture 2.1 also appears in the literature: as Question 3.1 of [Häg98], Conjecture 2.1 of [Häg03], and Conjecture 1.1 of [Lin11].

Conjecture 2.8. *Fix $p \in (0,1)$. Let G be a graph and $B = BB(G, V(G))$. Then, for all $x, y \in V(G)$, we have*

$$P(x_0 \leftrightarrow y_0 \text{ in } B_p) \geq P(x_0 \leftrightarrow y_1 \text{ in } B_p).$$

The case where $G = K_n$ and $p = 1/2$ has recently been proven [dB16]. We have not been able to show that Conjecture 2.8 is equivalent to the Bunk Bed Conjecture. Perhaps it is not.

2.2 Simulation

We now formalize the process of simulation in Lemmas 2.9 and 2.10. For notational convenience we will denote $G_{\mathbf{p}}$ by $G(\mathbf{p})$ in the following exposition.

Let G be a graph. For each $e \in E(G)$, let $(H_e, \mathbf{p}_e, x_e, y_e)$ be a 4-tuple where H_e is a graph, $\mathbf{p}_e : E(H_e) \to [0,1]$ is an edge-probability function on H_e, and x_e and y_e are distinct vertices of H_e. Let G' be the graph obtained from the vertex disjoint union $G'' = V(G) \sqcup \bigsqcup_{e \in E(G)} H_e$ by successively identifying vertices x and x_e and y and y_e for each edge $e = \{x, y\} \in E(G)$. The vertex resulting from the identifications of $x \in V(G)$ with other vertices in G'' will also be called x and we view $V(G)$ as a subset of $V(G')$. Likewise, each edge $f \in E(G')$ is viewed as belonging to the unique H_e from which it arose in G'. Let $\mathbf{p}' : E(G') \to [0,1]$ be defined by setting $\mathbf{p}'(f) = \mathbf{p}_e(f)$ if and only if $f \in E(H_e)$. Informally, G' is the graph obtained from G by replacing each edge e with H_e and $\mathbf{p}'$ is the edge-probability function on G' that restricts to $\mathbf{p}_e$ on H_e.

We define a map $\pi : \text{span}(G') \to \text{span}(G)$ as follows. If $H' \in \text{span}(G')$ then $H = \pi(H')$ is the spanning subgraph of G that retains $e = \{x_e, y_e\} \in E(G)$ if and only if $x_e \leftrightarrow y_e$ in $H' \cap H_e$. We define $\mathbf{q} : E(G) \to [0,1]$ by setting $\mathbf{q}(e) = P(x_e \leftrightarrow y_e \text{ in } H_e(\mathbf{p}_e))$ for $e \in E(G)$.

Lemma 2.9. *The following statements hold.*

1. *If $H \in$ span(G), then $P(G(\mathbf{q}) = H) = P(G'(\mathbf{p}') \in \pi^{-1}(H))$.*
2. *If $H' \in$ span(G') and $H = \pi(H')$, then, for all $x, y \in V(G)$, $x \leftrightarrow y$ in H' if and only if $x \leftrightarrow y$ in H.*
3. *For all $x, y \in V(G)$, $P(x \leftrightarrow y \text{ in } G(\mathbf{q})) = P(x \leftrightarrow y \text{ in } G'(\mathbf{p}'))$.*

We omit the proof of Lemma 2.9; the details are straightforward but tedious to verify.

Lemma 2.10. *Let G be a graph. For all $\epsilon > 0$ there is a $\gamma > 0$ such that for all $\mathbf{p}, \mathbf{q} : E(G) \to [0,1]$, if $|\mathbf{p}(e) - \mathbf{q}(e)| < \gamma$ for all $e \in E(G)$ then*

$$|P(G_{\mathbf{p}} \in A) - P(G_{\mathbf{q}} \in A)| < \epsilon$$

for all $A \subseteq$ span(G).

Proof. For each $A \subseteq \text{span}(G)$, $f_A(\mathbf{p}) = P(G_{\mathbf{p}} \in A)$ is a polynomial in the variables $\mathbf{p}(e)$. Thus f_A is continuous and even uniformly continuous for $\mathbf{p}$ in the compact set $[0,1]^{E(G)}$. Since G is finite, there are only finitely many $A \subseteq \text{span}(G)$, so given $\epsilon > 0$ we can find a bound $\gamma > 0$ that will apply uniformly to all f_A. □

2.3 Proof of the Main Theorem

We use the following class of 4-tuples as replacements in our proof of Theorem 2.6. For integers $j, k \geq 1$ and $p \in (0,1)$ we define $F_{j,k}(p) = (H_{j,k}, \mathbf{p}, h_0, h_k)$ where $H_{j,k}$ is the graph consisting of vertices h_0 and h_k and j internally vertex disjoint paths from h_0 to h_k, each of length k, and where $\mathbf{p}(e) = p$ for all $e \in E(H_{j,k})$. Theorem 2.6 is an immediate consequence of Lemma 2.11, listed below.

Lemma 2.11. *Let G be a graph, $T \subseteq V(G)$, and $B = BB(G,T)$. Let $\mathbf{p} : E(B) \to [0,1]$ be symmetric. Let $p \in (0,1)$ and $\epsilon > 0$. Then there is another bunk bed graph $B' = BB(G',T')$ with $V(G) \subseteq V(G')$ and $T' \subseteq V(G')$, such that for all $x, y \in V(G)$ and $i, j \in \{0,1\}$,*

$$|P(x_i \leftrightarrow y_j \text{ in } B_{\mathbf{p}}) - P(x_i \leftrightarrow y_j \text{ in } B'_p)| < \epsilon. \tag{2.1}$$

Proof. Fix $p \in (0,1)$ and $\epsilon > 0$. Let G be a graph, $T \subseteq V(G)$, and $B = BB(G,T)$. Let $\mathbf{p} : E(B) \to [0,1]$ be symmetric. We construct B' as follows.

For each edge $e \in E(G)$, we pick $j = j_e \geq 1$ and $k = k_e \geq 1$, and replace $e_0 \in E(B)$ by a copy of $F_{j,k}(p)$ in the upper bunk of B' and $e_1 \in E(B)$ by another copy of $F_{j,k}(p)$ in the lower bunk of B'. For each edge $e = \{x_0, x_1\} \in E(B)$ with $x \in T$ we pick $j = j_e \geq 1$ and $k = k_e \geq 1$ and replace e by $F_{j,2k+1}(p)$. This action adds j x_0-x_1 paths to B, each of length $2k+1$. For each such path $P = (x_0, h_1, \ldots, h_k, h_{k+1}, \ldots, h_{2k}, x_1)$, we view the sub-path $x_0 P h_k$ as belonging to the upper bunk of B', $h_{k+1} P x_1$ as belonging to the lower bunk of B', and the edge $\{h_k, h_{k+1}\}$ as being a post of B'. These new posts are the post edges of B'. See Figure 2.3.

Let $\mathbf{p}' : E(B') \to [0,1]$ and $\mathbf{q} : E(B) \to [0,1]$ be as in Lemma 2.9. Note that $\mathbf{p}'(e) = p$ for all $e \in E(B')$. Note that $\mathbf{q}$ is symmetric. By Lemma 2.10, we can pick $\gamma > 0$ so that if

$$|\mathbf{q}(e) - \mathbf{p}(e)| < \gamma \text{ for all } e \in E(B) \tag{2.2}$$

then (2.1) will hold. It is not hard to show that we can pick j_e and k_e so that (2.2) will hold. We can take each k_e to be odd and then each post edge e has $q(e) = 1 - (1 - p^{2k+1})^j$ for some $j, k \geq 1$. These numbers are dense in $[0,1]$.

□

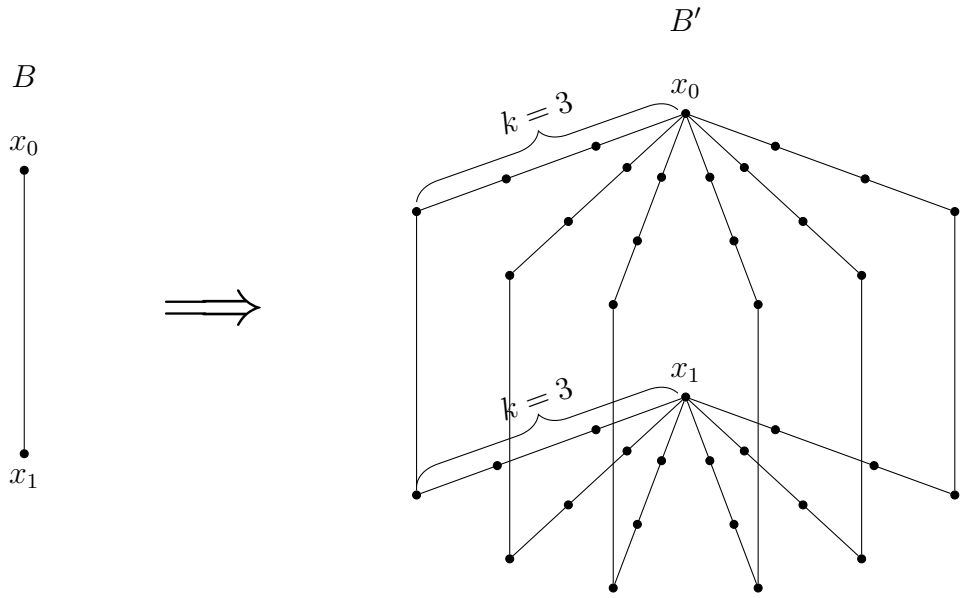

Figure 2.3. Post $\{x_0, x_1\}$ in B is replaced by $F_{6,7}$ and 6 posts in B'.

Proof. (of Theorem 2.6) Let G, T, B, and $\mathbf{p}$ be as in the statement of Conjecture 2.1. Fix $p \in (0,1)$. If we had

$$\delta = P(x_0 \leftrightarrow y_1 \text{ in } B_{\mathbf{p}}) - P(x_0 \leftrightarrow y_0 \text{ in } B_{\mathbf{p}}) > 0$$

for some $x, y \in V(G)$, then by applying Lemma 2.11 with $\epsilon = \delta/3$, we would have

$$P(x_0 \leftrightarrow y_1 \text{ in } B'_p) - P(x_0 \leftrightarrow y_0 \text{ in } B'_p)) > \delta/3 > 0,$$

contradicting the assumption of the theorem. □

Chapter 3: Monotone Boolean Function Testing

3.1 Introduction

If S is a set, let $\mathcal{P}(S) = \{A | A \subseteq S\}$ be its *power set*. A *Boolean function on* S is any function of the form $f : \mathcal{P}(S) \to \{0, 1\}$ for some set S. A Boolean function is *monotone* if for all subsets A and B of S, $A \subseteq B$ implies $f(A) \leq f(B)$.

In this chapter, we describe our new algorithms for computing the *ideal* $I(f)$ of a given monotone Boolean function, $f : \mathcal{P}(S) \to \{0, 1\}$, namely $I(f) = f^{-1}(0) = \{A \subseteq S : f(A) = 0\}$. By calculating the ideal $I(f)$, we mean listing all of its members. Variants of this problem include listing all the maximal elements of $I(f)$ or just computing its size, $|I(f)|$. Our algorithms can handle these variants as well.

Note that if f is monotone and $B \in I(f)$, i.e. $f(B) = 0$, then for all $A \subseteq B$, $f(A) = 0$ and $A \in I(f)$ as well. Thus $I(f)$ is closed under taking subsets and so is, by definition, an *abstract simplicial complex*. Thus our algorithms can list all the members of, or list all the maximal elements of, or calculate the size of any simplicial complex defined by a monotone Boolean function.

This problem originated from attempts to computationally test Conjecture 2.3. Let $G = (V, E)$ be a graph with vertex set V and edge set E. Let x and y be vertices of G. Define the Boolean function $f_{xy} : \mathcal{P}(E) \to \{0, 1\}$ such that for all subsets F of the edge set E we have $f_{xy}(F) = 1$ if and only if vertices x and y are connected in (V, F). It is easy to see that f_{xy} is monotone. This is because if a connection event holds true in (V, F) it must hold true in (V, F') for all $F' \supseteq F$; adding more edges cannot destroy connectivity. Conjecture 2.3 is thus about the number of subgraphs (V, F) of G for which $f_{xy}(F) = 1$, i.e. about the number $|f_{xy}^{-1}(1)| = 2^{|E|} - |f_{xy}^{-1}(0)|$. Conjecture 2.3 asserts $|f_{x_0y_0}^{-1}(1)| \geq |f_{x_0y_1}^{-1}(1)|$ for all x, y. As noted, we used our algorithms to test various cases of Conjecture 2.3.

Our algorithm for computing $I(f)$ compares very favorably with the existing algorithm due to Hansel [Han66], both in memory requirements and in parallelization. We focus on the number of computations of $f(A)$ for particular subsets A of S that

these algorithms need to perform as, in typical examples, such calculations are the ones that are most expensive to carry out. When given a monotone Boolean function f on an n-element set S, Hansel's algorithm requires memory space (i.e. number of memory locations) on the order of $2^n/\sqrt{n}$ and may need to evaluate f on the order of $2^n/\sqrt{n}$ subsets of S. In contrast, our algorithm requires space on the order of n^2 at the expense of only a logarithmic factor more function evaluations, on the order of $2^n \log(n)/\sqrt{n}$. The number of computations of f needed by any algorithm to list the elements of $I(f)$ can be on the order of $2^n/\sqrt{n}$ so all of these algorithms are optimal or nearly optimal in this sense.

The novel algorithm has the potential to finish computations even faster in some cases where "tree trimming" is possible, but this is dependent on the given monotone function. See Chapter 5 for a discussion of this.

In addition, our algorithm can be run in parallel whereas Hansel's algorithm cannot. If one has N processors available, the run time of the parallel version of our algorithm is on the order of $(1/N)2^n \log(n)/\sqrt{n}$ computations of f. With modern parallel clusters containing thousands of parallel processors, this yields a significant speed up. Note that as n increases, both algorithms will quickly become impossible to run due to time requirements that are exponential in n. But on a parallel cluster, our algorithm will be able to complete the analysis of many more small cases of n than Hansel's will be able to without running into space constraints. That significantly extends the limits of computational exploration of conjectures on monotone Boolean functions, including Conjecture 2.3.

The basis of our improved algorithm is our other new algorithm for producing the so-called *Christmas Tree Decomposition* of $\mathcal{P}(S)$ into symmetric chains. This will be explained in Section 3.5.

This chapter is organized as follows. In Section 3.2, we introduce Boolean functions and the main problems on Boolean functions that we will consider. In Sections 3.3 and 3.4 we will introduce some standard notation and the concept of symmetric chain decompositions. In Sections 3.5 and 3.6 we will introduce the Christmas Tree symmetric chain decomposition and give our new algorithms for computing it. We will use these algorithms to give algorithms for the main problems in Section 3.7. We analyze the performance of all of these algorithms in Section 3.8. We close with Section 3.9, where we will discuss how we used these algorithms to test various cases of the Bunk Bed Conjecture.

3.2 Boolean Functions and The Main Problems

If S is a set, let $\mathcal{P}(S) = \{A | A \subseteq S\}$ be its *power set*. A *Boolean function* on a finite set S is any function of the form $f : \mathcal{P}(S) \to \{0, 1\}$ for some set S. A Boolean function is *monotone* if for all subsets A and B of S, $A \subseteq B$ implies $f(A) \leq f(B)$.

If $f : \mathcal{P}(S) \to \{0,1\}$ is a monotone Boolean function, the *ideal* of f is the set $I(f) = f^{-1}(0) = \{A \subseteq S : f(A) = 0\}$. More generally, I is an *ideal* of subsets of a set S if whenever $B \in I$ and $A \subseteq B$ then $A \in I$, i.e. I is "closed under taking subsets." An *abstract simplicial complex* is defined to be a non-empty ideal of finite sets.

The *filter* of f is defined to be the set $F(f) = f^{-1}(1) = \mathcal{P}(\mathcal{S}) \setminus I(f) = \{A \subseteq S : f(A) = 1\}$. More generally, F is a *filter* of subsets of a set S if $A \in F$ and $A \subseteq B \subseteq S$ implies $B \in F$, i.e. F is "closed under taking supersets."

Suppose we are given an algorithm to compute a particular monotone Boolean function $f : \mathcal{P}(S) \to \{0,1\}$. We consider the following related problems.

Problem 3.1. List the elements of $I(f) = f^{-1}(0) = \{A \subseteq S : f(A) = 0\}$.

Problem 3.2. List the maximal elements of $I(f)$, i.e. the elements that are maximal with respect to inclusion.

Problem 3.3. Compute $|I(f)|$.

Problem 3.4. List the elements of $F(f) = f^{-1}(1) = \{A \subseteq S : f(A) = 1\}$.

Problem 3.5. List the minimal elements of $F(f)$, i.e. the elements that are minimal with respect to inclusion.

Problem 3.6. Compute $|F(f)|$.

We next discuss some interesting classes of monotone Boolean functions f and their ideals and filters. In particular, we consider those arising from computationally defined abstract simplicial complexes and those that arise from monotone graph properties. This will demonstrate that Problems 3.1-3.6 are natural questions in those contexts.

We will first need the following theorem on how monotone Boolean functions give rise to abstract simplicial complexes and filters and vice versa.

Lemma 3.7. *We have the following statements.*

1. *Suppose S is a finite set and $f : \mathcal{P}(S) \to \{0,1\}$ is a monotone Boolean function with $f(A_0) = 0$ for some $A_0 \subseteq S$. Then $I(f)$ is an abstract simplicial complex of subsets of S.*

2. *Suppose Δ is an abstract simplicial complex of subsets of a finite set S. Suppose also that $f : \mathcal{P}(S) \to \{0,1\}$ is defined by setting $f(A) = 0$ for all $A \in \Delta$ and $f(A) = 1$ for all $A \notin \Delta$. Then f is a monotone Boolean function with $f(A_0) = 0$ for some $A_0 \subseteq S$. Also $I(f) = \Delta$.*

3. *Suppose S is a finite set and $f : \mathcal{P}(S) \to \{0,1\}$ is a monotone Boolean function. Then $F(f)$ is a filter of subsets of S.*

4. *Suppose F is a filter of subsets of a finite set S. Suppose also that $f : \mathcal{P}(S) \to \{0,1\}$ is defined by setting $f(A) = 0$ for all $A \notin F$ and $f(A) = 1$ for all $A \in F$. Then f is a monotone Boolean function. Also $F(f) = F$.*

Proof. Proof of statement 1. We have that $I(f)$ is not empty. The set A_0 is in $I(f)$ since $f(A_0) = 0$. Suppose now that $A \subseteq B \in I(f)$. Since f is monotone, we have $0 \leq f(A) \leq f(B) = 0$ and $f(A) = 0$ as well. So $A \in I(f)$ and $I(f)$ is closed under taking subsets. Thus $I(f)$ satisfies all the requirements to be an abstract simplicial complex.

Proof of statement 2. By definition, $I(f) = \Delta$. Since Δ is not empty, it contains some set A_0. We have $f(A_0) = 0$. The function f must be monotone. If it were not, we would have $A \subseteq B \subseteq S$ such that $f(A) \not\leq f(B)$. Then $f(A) = 1$ and $f(B) = 0$ and we have $A \subseteq B \in \Delta$ with $A \notin \Delta$. This contradicts the assumption that Δ is a simplicial complex. We have proved that f satisfies all the stated conditions.

Proof of statement 3. Suppose $A \in F(f)$ and $A \subseteq B \subseteq S$. Since f is monotone, we have $1 = f(A) \leq f(B) \leq 1$. Thus $f(B) = 1$ and $B \in F(f)$ as well. Thus $F(f)$ is closed under taking supersets and is hence a filter.

Proof of statement 4. By definition, $F(f) = F$. Also, the function f must be monotone. If it was not, then there would be sets $A \subseteq B \subseteq S$ such that $f(A) \not\leq f(B)$. This means that $f(A) = 1$ and $f(B) = 0$ or $A \in F$ and $B \notin F$. This contradicts the assumption that F is a filter. □

We say an abstract simplicial complex Δ of subsets of a finite set S is *computationally defined* if there is some algorithm that, when given $A \subseteq S$ as input, decides whether $A \in \Delta$ or not. Given such a simplicial complex Δ, define a Boolean function $f : \mathcal{P}(S) \to \{0,1\}$ by $f(A) = 0$ if and only if the algorithm determines that $A \in \Delta$. By part 2 of Lemma 3.7, this f is monotone and $I(f) = \Delta$. Then Problem 3.1, listing all the simplices of $I(f) = \Delta$, Problem 3.2, listing all the maximal simplices of $I(f) = \Delta$, and Problem 3.3, computing the size of $|I(f)| = |\Delta|$, are all natural and important questions.

Here is an example of a simplicial complex to which we can apply our algorithms. Given a collection of balls $\mathcal{B} = \{B_i : i \in S\}$ in $\mathbb{R}^d$, the *Čech complex* of $\mathcal{B}$ is the simplicial complex $\check{\text{C}}\text{ech}(\mathcal{B}) = \{A \subseteq S : \bigcap_{i \in A} B_i \neq \emptyset\}$. The Čech complex is a central object of study in the rapidly growing field of topological data analysis [EH10]. $\check{\text{C}}\text{ech}(\mathcal{B})$ is an abstract simplicial complex. Clearly if a set of balls has non-empty intersection, then so does any subset of that set of balls and so $\check{\text{C}}\text{ech}(\mathcal{B})$ is closed under taking subsets. Here we adopt the typical convention that an empty intersection is the whole space, i.e. $\bigcap_{i \in \emptyset} B_i = \mathbb{R}^d$. In particular, this intersection is considered not empty and so $\emptyset \in \check{\text{C}}\text{ech}(\mathcal{B})$ and the complex is not empty. Note that reducing the number of computations of $\bigcap_{i \in A} B_i$ is important as these are typically computationally expensive tasks to carry out. Our algorithms perform an optimal number of such calculations.

We say a filter F of subsets of a finite set S is *computationally defined* if there is some algorithm that, when given $A \subseteq S$ as input, decides whether $A \in F$ or not. Given such a filter f, define a Boolean function $f : \mathcal{P}(S) \to \{0,1\}$ by $f(A) = 0$ if and only if $A \notin F$ and $f(A) = 1$ if $A \in F$. By part 4 of Lemma 3.7, this f is monotone and $F(f) = F$. Then Problem 3.4, listing all the elements of $F(f) = F$, Problem 3.5, listing all the minimal elements of $F(f) = F$, and Problem 3.6, computing the size of $|F(f)| = |F|$, are all natural and important questions.

We will now give many examples of filters. We first have to define a monotone property of graphs. Given a set V of "vertices", let $S = \{\{v, w\} : v, w \in V\}$ the set of all possible "edges" on V. The set of all graphs (V, E) with vertex set V can thus be viewed as $\mathcal{P}(S)$. Since the vertex set V is the same for all graphs, each such graph is uniquely determined by its edge set $E \subseteq S$ and every $E \subseteq S$ gives rise to a unique graph (V, E). A property P of graphs is said to be *monotone* if for every graph $G = (V, E)$ that satisfies property P, we have that every graph $H = (V, E')$ with $E' \supseteq E$ also satisfies P. In other words, a graph property is monotone if "it is preserved by adding edges." Let $F = \{E \subseteq S : (V, E) \text{ satisfies property } P\}$. Since P is monotone, we see that F is a filter of subsets of S. Define $f : \mathcal{P}(S) \to \{0,1\}$ by setting $f(E) = 1$ if and only if (V, E) satisfies property P and by setting $f(E) = 0$ if and only if (V, E) does not satisfy property P. Then part 4 of Lemma 3.7 shows that f is a monotone Boolean function and $F(f)$ is the filter of all graphs (V, E) that satisfy property P.

Dozens of interesting and well-studied properties of graphs are monotone. For example, the properties that G is connected, that G contains a particular fixed type of subgraph, or that the chromatic number of G is larger than some threshold k are all monotone. Indeed, adding edges preserves all of these properties. So all of these properties P give rise to monotone Boolean functions f for which the corresponding filter $F(f)$ is the set of graphs satisfying P. In this setting, Problem 3.4, i.e. listing all graphs with property P (those in $F(f)$), Problem 3.5, listing the minimal graphs with property P, and Problem 3.6, counting all graphs with property P, are all natural and important questions.

In the rest of this chapter, we will restrict our attention to Problems 3.1-3.3. Indeed we shall now show that algorithms for these problems can be re-purposed to solve Problems 3.4-3.6.

We will first need the following definitions and lemma.

Suppose a set S and a Boolean function $f : \mathcal{P}(S) \to \{0,1\}$ are given. Define $A^c = S \setminus A$ for all $A \subseteq S$. Also define the Boolean function $g : \mathcal{P}(S) \to \{0,1\}$ that is *complementary* to f by setting $g(A) = 1 - f(A^c)$ for all $A \subseteq S$.

Lemma 3.8. *Suppose S is a finite set and $f : \mathcal{P}(S) \to \{0,1\}$ is a monotone Boolean function. Let g be the complementary Boolean function corresponding to f. Then we have the following statements.*

1. *Any algorithm to compute f can be re-purposed to compute g.*

2. *g is a monotone Boolean function.*

3. *$F(f) = \{A^c : A \in I(g)\}$.*

4. *A^c is a minimal element of $F(f)$ if and only if A is a maximal element of $I(g)$.*

5. *$|F(f)| = |I(g)|$.*

Proof. Statement 1 is a triviality. If you have an algorithm that can compute $f(A)$ for any $A \subseteq S$, then that same algorithm can be used to compute $g(A) = 1 - f(A^c)$ for any $A \subseteq S$.

Proof of statement 2. Suppose $A \subseteq B \subseteq S$. Then $B^c \subseteq A^c$ and $f(B^c) \leq f(A^c)$ since f is monotone. But then $g(A) = 1 - f(A^c) \leq 1 - f(B^c) = g(B)$. This shows that g is monotone.

Proof of statement 3. We have $F(f) = \{A \subseteq S : f(A) = 1\}$. Re-indexing, we get $F(f) = \{A^c \subseteq S : f(A^c) = 1\} = \{A^c \subseteq S : g(A) = 1 - f(A^c) = 0\} = \{A^c \subseteq S : A \in I(g)\}$.

Proof of statement 4. Suppose now that A^c is a minimal element of $F(f)$. By definition of minimality, this is equivalent to the statement that $f(A^c) = 1$ and $f(B^c) = 0$ for all $B^c \subsetneq A^c$. But this is equivalent to the statement that $g(A) = 1 - f(A^c) = 0$ and $g(B) = 1 - f(B^c) = 1$ for all $B \supsetneq A$. By the definition of maximality, this is equivalent to the statement that A is a maximal element of $I(g)$.

Statement 5 follows trivially from Statement 3. □

Theorem 3.9. *Let S be a finite set and let $f : \mathcal{P}(S) \to \{0, 1\}$ be a monotone Boolean function. Suppose we have an algorithm that can compute $f(A)$ for any $A \subseteq S$. Then we have the following statements.*

1. *Any algorithm to solve Problem 3.1 can be re-purposed to solve Problem 3.4.*

2. *Any algorithm to solve Problem 3.2 can be re-purposed to solve Problem 3.5*

3. *Any algorithm to solve Problem 3.3 can be re-purposed to solve Problem 3.6*

Proof. By part 1 of Lemma 3.8, the algorithm for computing f gives us an algorithm to compute g. By part 2 of Lemma 3.8, g is monotone.

Suppose we have an algorithm to solve Problem 3.1. Then the following is an algorithm to solve Problem 3.4. Given f, use the algorithm to solve Problem 3.1 to list the elements A of $I(g)$. However, as you go, list the complementary sets A^c instead. By part 3 of Lemma 3.8, this lists the elements of $F(f) = \{A^c : A \in I(g)\}$.

Suppose we have an algorithm to solve Problem 3.2. Then the following is an algorithm to solve Problem 3.5. Given f, use the algorithm to solve Problem 3.2 to

list the maximal elements A of $I(g)$. However, as you go, list the complementary sets A^c instead. By part 4 of Lemma 3.8 this lists the minimal elements of $F(f)$.

Suppose we have an algorithm to solve Problem 3.3. Then the following is an algorithm to solve Problem 3.6. Given f, use the algorithm to solve Problem 3.3 to compute $|I(g)|$. By part 4 of Lemma 3.8, this also computes $|F(f)| = |I(g)|$.

□

Our goals in producing algorithms for Problems 3.1-3.3 (and hence, by Theorem 3.9, also for Problem 3.4-3.6) is to reduce the memory space required and time required as much as possible. To measure the time our algorithms take, we focus on the number of computations of f that the algorithms need to make. Typically, these computations are the ones that are most time-consuming. Calculating whether a collection of balls in $\mathbb{R}^d$ have a non-empty intersection is a computationally expensive task and so the monotone Boolean function that computes the Čech complex is hard to compute. Deciding whether a graph satisfies the monotone property of being at least 4-chromatic is an NP-complete problem so there is no known efficient algorithm for computing the corresponding monotone Boolean function [GJ79]. Indeed it is widely believed that no efficient algorithm exists for this problem. Many monotone properties besides this one example are NP-complete and thus hard to compute.

We shall see in Section 3.8 that if we are given an arbitrary Boolean function $f : \mathcal{P}(S) \to \{0, 1\}$ on an n-element set S, then any algorithm to solve one of Problems 3.1 or 3.3 must, in the worst case, evaluate $f(A)$ on all 2^n subsets A of S. There we shall also see that if f is monotone, there are algorithms that can solve Problems 3.1-3.3 slightly more efficiently, i.e. taking on the order of $2^n/\sqrt{n}$ computations of f. Remarkably, we will also see there that this seemingly mild improvement is in fact optimal.

3.3 Correspondences Between Sets and 0-1 Vectors

In the rest of this chapter, we will identify subsets $A \subseteq S$ of an n element set S with n-dimensional 0-1 valued vectors in $\{0, 1\}^n$. We will also identify Boolean functions $f : \mathcal{P}(S) \to \{0, 1\}$ with n variable functions $f : \{0, 1\}^n \to \{0, 1\}$. The identifications we adopt are all standard in discrete mathematics but, for the sake of completeness, we define them all here.

If n is a positive integer, the *Boolean lattice* is $B_n = \{0, 1\}^n$. That is, B_n is the set of all n-tuples with all entries either 0 or 1. We often write such elements as strings, e.g. we write $(0, 1, 1, 0)$ as 0110 and so on. There is a simple bijection between B_n and $\mathcal{P}(S)$ where $S = \{s_1, \ldots, s_n\}$ is an n-element set with its elements indexed in some definite fixed order. If $A \subseteq S$, we identify A with its indicator vector $x_A = (x_1, \ldots, x_n) \in B_n$ where $x_i = 1$ if $s_i \in A$ and $x_i = 0$ if $s_i \notin A$. Thus the set $A = \{b, c\} \subseteq S = \{a, b, c, d\}$

would be identified with the indicator $x_A = (0, 1, 1, 0) \in B_4$ or $x_A = 0110$. Note that $x_\emptyset$ is the all 0's string, $x_\emptyset = 00\ldots0$, and x_S is the all 1's string, $x_S = 11\ldots1$.

The *Hamming weight* or just *weight* of a string $x \in B_n$ is defined to be $|x| = \sum_i x_i$. Note that $|x|$ is "the number of 1's in the string x," i.e. the number of i such that $x_i = 1$ or $i \in A$. Thus we have $|x_A| = |A|$.

We put the following partial order on B_n. If $x = (x_1, \ldots, x_n)$ and $y = (y_1, \ldots, y_n)$, we set $x \leq y$ if and only if $x_i \leq y_i$ for all i. With this definition, we see that that if $A, B \subseteq S$, then $x_A \leq x_B$ if and only if $A \subseteq B$. Indeed, the following statements are equivalent: (i) $x_A \leq x_B$, (ii) for all i, $(x_A)_i = 1$ implies $(x_B)_i = 1$, (iii) for all i, $i \in A$ implies $i \in B$, and (iv) $A \subseteq B$.

From now onwards, all of these identifications will be understood. We will write $\mathcal{P}(S)$ interchangeably with B_n, $A \subseteq S$ interchangeably with $x_A = (x_1, \ldots x_n) = x_1 x_2 \ldots x_n$, and $|A|$ interchangeably with $|x_A|$.

If $f : \mathcal{P}(S) \to \{0, 1\}$ is a Boolean function then we define a corresponding function $\tilde{f} : B_n \to \{0, 1\}$ by setting $\tilde{f}(x_A) = f(A)$ for all $A \subseteq S$. For example, suppose $f : \mathcal{P}(S) \to \{0, 1\}$ with $S = \{a, b\}$ is defined by setting $f(\{a, b\}) = 1$ and $f(\emptyset) = f(\{a\}) = f(\{b\}) = 0$. Then $\tilde{f} : \{0, 1\}^2 \to \{0, 1\}$ has $\tilde{f}(11) = 1$ and $\tilde{f}(00) = \tilde{f}(10) = \tilde{f}(01) = 0$.

We say a Boolean function $f : B_n \to \{0, 1\}$ is *monotone* if for all $x, y \in B_n$, $x \leq y$ implies $f(x) \leq f(y)$. Thus a Boolean function $f : \mathcal{P}(S) \to \{0, 1\}$ is monotone if and only if the corresponding Boolean function $\tilde{f} : B_n \to \{0, 1\}$ is monotone. Indeed, the statement that $A \subseteq B$ implies $f(A) \leq f(B)$ is equivalent to the statement that $x_A \leq x_B$ implies $\tilde{f}(x_A) \leq \tilde{f}(x_B)$.

With these correspondences in mind, we will also call functions $\tilde{f} : B_n \to \{0, 1\}$ *Boolean functions* and identify $f : \mathcal{P}(S) \to \{0, 1\}$ with its corresponding $\tilde{f} : B_n \to \{0, 1\}$. We will also interchangeably say f is *monotone* if $f(x) \leq f(y)$ for $x \leq y$ in B_n or $f(A) \leq f(B)$ for $A \subseteq B \subseteq S$.

3.4 Symmetric Chain Decompositions

Here we define a *symmetric chain decomposition* of a partially ordered set. In order to do this, we will first have to define many other terms, e.g. poset, rank, chains, saturated chains, etc.

If P is a set and $\leq$ is partial order on P, we say the pair $(P, \leq)$ is a *partially ordered set* or *poset* for short. We will just write P for the poset if the partial ordering $\leq$ is understood from the context. We write $a < b$ if $a \leq b$ and $a \neq b$.

If $(P, \leq_P)$ is any poset then then any subset Q of P determines another poset $(Q, \leq_Q)$ where $\leq_Q$ is the restriction of $\leq_P$ to the domain $Q \times Q$. We say any such Q is a *subposet* of P. An *isomorphism* of posets P and Q is any *order-preserving bijection*, i.e. a bijection $f : P \to Q$ such that $x \leq y$ in P if and only if $f(x) \leq f(y)$ in Q. We

say a poset $(P, \leq_P)$ contains a *copy* of another poset $(Q, \leq_Q)$ if and only if there is a subposet of P that is isomorphic to Q.

We say two elements a and b in a poset are *comparable* if $a \leq b$ or $b \leq a$. Otherwise, we say the elements are *incomparable*. A poset is *totally ordered* if every pair of elements in it are comparable. If k is an integer with $k \geq 0$ then a length k *chain* is any poset that is isomorphic to the totally ordered poset $C_k = \{0, 1, 2, \ldots, k\}$ under the usual ordering of the integers. A size k *antichain* is any poset isomorphic to the poset $A_k = \{1, 2, \ldots, k\}$ with the empty partial order, i.e. with i and j incomparable for all $i \neq j$. Clearly any sequence of elements $x_0 < x_1 < \ldots < x_k$ in a poset P is isomorphic to C_k so we call such a sequence a *chain* of P. Similarly any subset $\{y_1, \ldots, y_k\}$ of pairwise incomparable elements of a poset P is isomorphic to A_k, so we call such a set an *antichain* of P.

The set $B_n = \{0, 1\}^n$ with its partial order, namely $x \leq y$ if and only if $x_i \leq y_i$ for all i, is a partially ordered set that is isomorphic to the partially ordered set $\mathcal{P}(S)$ with its partial order, namely $A \leq B$ if and only if $A \subseteq B$. The length k chains of these posets correspond to sequences of subsets with $A_0 \subsetneq A_1 \subsetneq A_2 \subsetneq \cdots \subsetneq A_k$ and the size k antichains in these posets correspond to the sets $\{A_1, \ldots, A_k\}$ of subsets A_i with no A_i contained in any other A_j with $j \neq i$. The set $L(S, k) = \{A \subseteq S : |A| = k\}$ consisting of all size k subsets of S is an antichain with $\binom{n}{k}$ elements. Similarly, the corresponding set $L(n, k) = \{x \in B_n : |x| = k\}$ in B_n consisting of all the weight k strings in B_n is an antichain, also with $\binom{n}{k}$ elements.

If a and b are two elements in a poset $(P, \leq)$ we say b *covers* a or a is *covered* by b if $a < b$ and there is no element $z \in P$ such that $a < z < b$. We denote this by $a \lessdot b$. In B_n we have $x \lessdot y$ if and only if there is exactly one index i such that $x_i = 0$ and $y_i = 1$, i.e. if "x has only one more 1 than y". Analogously we have $A \lessdot B$ in $\mathcal{P}(S)$ if and only if $B = A \cup \{i\}$ for some $i \notin A$, i.e. "B has one more element than A" .

A chain $c_0 < c_1 < \cdots < c_k$ in a poset is *saturated* if and only if the $<$ relations are also covering relations, $c_0 \lessdot c_1 \lessdot c_2 \lessdot \cdots \lessdot c_k$. So a chain $c_0 < c_1 < \cdots < c_k$ in B_n is saturated if and only if the weights $|c_i|$ satisfy $|c_i| = |c_{i-1}| + 1$ for all $1 \leq i \leq k$. Similarly a chain $A_0 \subsetneq A_1 \subsetneq A_2 \subsetneq \ldots \subsetneq A_k$ in $\mathcal{P}(S)$ is saturated if and only if $|A_i| = |A_{i-1}| + 1$ for all $1 \leq i \leq k$.

If there is an element $\hat{0} \in P$ such that $x \geq \hat{0}$ for all $x \in P$, then we say $\hat{0}$ is the *zero-element* of P. Note that if a zero-element exists, it is unique because if a and b are both zero elements, then $a \geq b$ and $b \geq a$ and so $a = b$. Similarly, if there is an element $\hat{1} \in P$ such that $x \leq \hat{1}$ for all $x \in P$, then we say $\hat{1}$ is the *top element*. If it exists, $\hat{1}$ is also unique. In B_n, $\hat{0} = 00\ldots0$ and $\hat{1} = 11\ldots1$. Correspondingly, $\hat{0} = \emptyset$ and $\hat{1} = S$ in $\mathcal{P}(S)$.

We say a poset is *ranked* if it has a zero element $\hat{0}$ and the property that for any $x < y$, all saturated chains from x to y have the same length. In a ranked poset P, we define the *rank function* $r : P \to \mathbb{N}$ by setting $r(x)$ to be the common length of

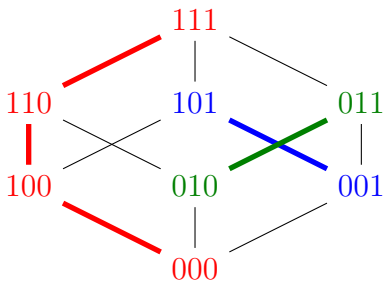

Figure 3.1. An SCD of B_3, namely $CTD(3)$.

Chain 1	Chain 2	Chain 3
111		
110	011	101
100	**01$\underline{0}$**	**$\underline{0}$01**
$\underline{000}$		

Figure 3.2. A SCD of B_3, namely $CTD(3)$. The minimal string of each chain is in bold. The unmatched 0's in the minimal strings are underlined.

each saturated chain from $\hat{0}$ to x. If P is a finite ranked poset, then the *rank* of P is the maximum rank of any element in P, $r(P) = \max_{x \in P} r(x)$.

The Boolean lattice, B_n, is a ranked poset with rank function $r(x) = |x|$ and the isomorphic poset $\mathcal{P}(S)$ (for $|S| = n$) has rank function $r(A) = |A|$. The rank of these posets is n, $r(B_n) = r(\mathcal{P}(S)) = n$.

If P is a ranked poset, a *symmetric chain* is a saturated chain $x_0 \lessdot \cdots \lessdot x_k$ with $r(x_0) + r(x_k) = r(P)$. The rationale behind this name is that the ranks of the elements in the chain will be symmetrically distributed about $r(P)/2$. Namely, a symmetric chain will have an element x with $r(x) = r(P)/2 - \ell/2$ if and only if it it has an element y with $r(y) = r(P)/2 + \ell/2$. Indeed, if $x = x_i$, then $y = x_{k-i}$ as we will see now. Note that since the chain is saturated, $r(x_i) = r(x_0) + i$ and $r(x_{k-i}) = r(x_k) - i$ for all $0 \leq i \leq k$. Thus $r(x_i) + r(x_{k-i}) = (r(x_0) + i) + (r(x_k) - i) = r(x_0) + r(x_k) = r(P)$ for all $0 \leq i \leq k$. Since $r(x_i) + r(x_{k-i}) = r(P)$, if $r(x_i) = r(P)/2 - \ell/2$, then $r(x_{k-i}) = r(P)/2 + \ell/2$.

A *symmetric chain decomposition*, or *SCD*, of a ranked poset is a partition of the elements of P into symmetric chains. See Figure 3.1 for an SCD of B_3 and Figure 3.2 for the elements of the chains in that SCD. See Figure 3.3 for the elements of the chains of an SCD for B_4. Note that B_3 is displayed in Figure 3.2 via a *Hasse diagram*, one in which an edge is directed upwards from vertex x to vertex y if and only if $x \lessdot y$.

Chain 1	Chain 2	Chain 3	Chain 4	Chain 5	Chain 6
1111					
1110	0111	1011	1101		
1100	0110	1010	1001	**0101**	**0011**
1000	**0100**	**0010**	**0001**		
0000					

Figure 3.3. An SCD of B_4, namely $CTD(4)$. The minimal string of each chain is in bold. The unmatched 0's in the minimal strings are underlined.

3.5 The Christmas Tree Decomposition

Theorem 3.10. *For all positive integers n, the poset B_n has a symmetric chain decomposition. (Alternatively, for each finite set S, $\mathcal{P}(S)$ has a symmetric chain decomposition.)*

This theorem was first proven in [dBvETK51] via the construction of a specific symmetric chain decomposition, $CTD(n)$, the so-called *Christmas Tree Decomposition*. The reason for this fanciful name is that when the elements of its chains are listed in horizontal rows that are centered on the page, the shape of the text looks a little like a silhouette of a Christmas tree; the horizontal lengths of the chains are long and short in a seemingly organic, unpredictable way.

We now give the construction of $CTD(n)$ as developed in [GK76]. We closely follow the presentation of this construction in [Jor10]. For each binary sequence $x = x_1 \dots x_n \in B_n$, we perform a so-called bracketing or parenthesis matching procedure as follows. We represent each 0 with a left parenthesis "(" and each 1 with a right parenthesis ")". We scan through the characters of the string starting at the left. Whenever we encounter a 0, i.e. a (, it becomes "unmatched". Whenever a 1, i.e. a), is encountered, it is matched to the rightmost unmatched 0, and this 1 now becomes matched as well. If there are currently no unmatched 0's, then this 1 is unmatched. We continue in this manner until we reach the end of the string on the right. This process matches a (to a) in the parenthesis string in the same way that a mathematician scanning the string would naturally try to match parentheses.

We now have three sets associated with the given sequence x, the set of positions of unmatched 0's, $U_0(x)$, the set of positions of unmatched 1's, $U_1(x)$, and the set of matchings $M(x) = \{(a, b) : \text{ a 0 in position } a \text{ is matched to 1 in position } b\}$.

$$x = 1011011100010110$$

$$\text{parentheses string for } x =)())()))(((()())($$

$$U_0(x) = \{9, 16\} \quad U_1(x) = \{1, 4, 7, 8\}$$

$$M(x) = \{(2,3),(5,6),(10,15),(11,12),(13,14)\}$$

Note if $a \in U_1(x)$ and $b \in U_0(x)$ then $a < b$. In words, all unmatched ones precede all unmatched zeros.

We next introduce a function τ which acts on the strings by changing the leftmost unmatched 0 to a 1. The function τ is defined on all $x \in B_n$ such that $U_0(x) \neq \emptyset$. For $x \in B_n$ with $|U_0(x)| = k$, let $C_x = \{x, \tau(x), \tau^2(x), \ldots, \tau^k(x)\}$. The *Christmas Tree Decomposition* is

$$CTD(n) = \{C_x : x \in B_n, U_1(x) = \emptyset\}.$$

Figures 3.2 and 3.3 exhibit $CTD(n)$ for B_n for $n = 3$ and $n = 4$, respectively. The minimal elements of the chains, those x with no unmatched 1's (i.e. with $U_1(x) = \emptyset$) are listed in bold face. The positions of the unmatched 0's, i.e. the positions in $U_0(x)$, are underlined. Notice how the successive elements of each chain C_x are formed, the unmatched 0's are flipped one by one to 1's proceeding from left to right.

Lemma 3.11. *The number of chains in $CTD(n)$ is $M_n = \binom{n}{\lfloor n/2 \rfloor}$.*

Proof. Let $k = \lfloor n/2 \rfloor$. Consider the level $L(n,k) = \{x \in B_n : r(x) = |x| = k\}$. We will show that there is a one-to-one correspondence between each chain C in $CTD(n)$ and the unique element x in $L(n,k)$ that is contained within it. This then shows that the number of chains in $CTD(n)$ is $M_n = \binom{n}{k} = |L(n,k))|$.

Since the chains in $CTD(n)$ are symmetric chains in B_n, they each must contain an element x with rank $r(x) = n/2$ if n is even, or elements x and y of ranks $r(x) = \lfloor n/2 \rfloor$ and $r(y) = \lceil n/2 \rceil$ if n is odd. Thus they each contain at least one element x of $L(n,k)$. Since the chains partition B_n, each element x in $L(n,k)$ must be in exactly one chain in $CTD(n)$. $L(n,k) = \{x \in B_n : r(x) = |x| = k\}$ is an antichain in B_n, i.e. contains no two comparable elements. Since every pair of elements in a chain is comparable, no chain can contain more than one element x of $L(n,k)$. Thus the given correspondence between $CTD(n)$ and $L(n,k)$ is a bijection, as claimed. □

3.6 Our Christmas Tree Decomposition Algorithms

3.6.1 The Tree T_n on the Minimal Elements of $CTD(n)$

Let $S_n = \{x \in B_n : U_1(x) = \emptyset\}$ be the set of the minimal elements of the chains in $CTD(n)$. We will also sometimes refer to minimal elements as initial strings. Let x_0 be the all zero string $\hat{0}$. Define $\sigma : S_n \setminus \{x_0\} \to S_n$ so that $\sigma(x)$ is obtained from the string x by changing the leftmost matched 1 of x to a 0. We define a tree $T_n = (S_n, E)$ on the vertex set S_n by letting $E = \{(x, \sigma(x)) : x \in S \setminus \{x_0\}\}$. The tree T_4 is exhibited in Figure 3.4. The elements of S_4, the minimal elements of $CTD(4)$, are shown as the

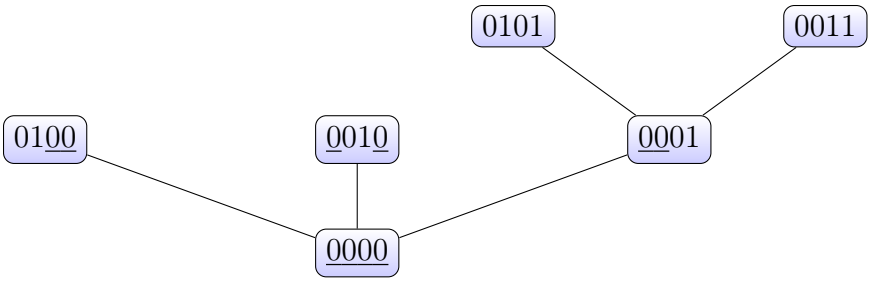

Figure 3.4. The tree T_4 of the minimal elements of $CTD(4)$.

vertices and the unmatched 0's within them are underlined. For each $x \neq x_0$, $\sigma(x)$ is drawn above x.

We show that T_n is a *tree* in the sense of graph theory, namely it is a connected graph that contains no cycles. Clearly, given any vertex x in T_n, $\sigma^k(x)$ will be the all 0's string x_0 when $k = |x|$, as each application of σ removes a 1. Thus, every x in T_n is connected by a path $x, \sigma(x), \sigma^s(x), \ldots, \sigma^k(x) = x_0$ to x_0. Thus T_n is *connected* as every vertex can be connected to every other through x_0. To see that T_n contains no cycle, suppose instead that it does contain a cycle C. C must contain a maximal weight vertex x, say $|x| = k$. Since C is a cycle, there must be two distinct neighbors y_0 and y_1 of x on the cycle. Since y has maximal weight and edges in T_n only connect vertices whose weights differ by 1, we have $|y_0| = |y_1| = k - 1$. But there is only one vertex y with weight $|y| \leq k - 1$ that is a neighbor of x, namely $y = \sigma(x)$. This is a contradiction.

A *rooted tree* is a tree with a distinguished vertex r called the *root* of T. If x is a vertex in a rooted tree T with root r and $x \neq r$ then there is a unique path in T from x to r. We say every vertex y on this path is an *ancestor* of x and that x is a *descendant* of each such vertex y. The ancestor y of x that is adjacent to x is called the *parent* of x and x is called a *child* of y.

We view T_n as a rooted tree with root $r = x_0$. If $x \neq x_0$ then the unique path from x to x_0 in T_n is $x, \sigma(x), \sigma^2(x), \ldots, \sigma^k(x) = x_0$ where $k = |x|$. The vertices $\sigma^i(x)$ are the ancestors of x, $\sigma(x)$ is the parent of x, and x is the child of $\sigma(x)$. Thus we see that the tree T_4 is drawn in Figure 3.4 with the progenitor of the family, the root x_0, written at the bottom and parents drawn directly below their children and the later descendants drawn above those children.

Every vertex x of T_n has $U_1(x) = \emptyset$ and so is a minimal element of a chain $C_x \in CTD(n)$, namely $C_x = \{x, \tau(x), \tau^2(x), \ldots, \tau^\ell(x)\}$ with $\ell = |U_0(x)|$. Thus we can extend T_n to a tree T'_n with vertex set B_n as follows. For each $x \in T_n$ we add a path $x, \tau(x), \tau^2(x), \ldots, \tau^\ell(x)$ with $\ell = |U_0(x)|$. This tree has all the vertices of B_n because $CTD(n)$ is a partition of B_n. Additionally, the tree T'_n has the important property

that every vertex x in B_n can be reached from x_0 by a path in T'_n, $x_0, x_1, \ldots, x_m = x$ in which each x_{i+1} has one more 1 than its parent x_i.

3.6.2 Our Recursive Algorithm to Produce $CTD(n)$

We can now define our novel recursive algorithm, *Recursive-CTD*, that visits every chain in $CTD(n)$ and every vertex of B_n. More generally, if it is given a vertex $x \in S_n$ as an input, it will visit x and every descendant of x in T'_n.

For any string x in B_n, let z_x denote the number of zeros in the first consecutive block of zeros in the string. Any string starting with a 1 would have $z_x = 0$. Since each $x \in S_n$ has no unmatched ones, we have $z_x > 0$ for all $x \in S_n$. Also let u_x denote the number of unmatched zeros within the first block of zeros, with $0 \leq u_x \leq z_x$. Note by the matching procedure all of the unmatched zeros must occur at the beginning of the block of zeros. Also $u_x = z_x$ only when $x = x_0$ (when x is the all zero string). As before $U_0(x)$ is the set of indices of all unmatched zeros in x. With these definitions, we can now describe the recursive algorithm.

Algorithm 3.12. *Algorithm: Recursive-CTD*

Input: string $x \in S_n$ and optional arguments z_x, u_x, $U_0(x)$

Output: none, recursive procedure to visit every descendant of x in T'_n

1. Calculate z_x, u_x, $U_0(x)$ if they are not provided as input.
2. Visit each element of $C_x = \{x, \tau(x), \ldots, \tau^k(x)\}$ where $k = |U_0(x)|$ switching the 0's in positions $j \in U_0(x)$ to 1's sweeping left to right.
3. If $u_x \leq 1$, then return, i.e. end the current function call.
4. For $i \in \{2, \ldots, z_x\}$ do
   (a) child $\leftarrow x$
   (b) child[i] $\leftarrow 1$
   (c) $m \leftarrow \min(i, u_x)$
   (d) Call Recursive-CTD(child, $i-1, m-2, U_0(x) \setminus \{m, m-1\}$).
5. End for.
6. Return.

Before we discuss the correctness of the algorithm we will need a lemma. Define the density of a string x, denoted $\delta(x)$, to be the weight of the string divided by the string length n (this amounts to the "proportion" of 1's in the string). Now define the max density of a string, denoted $\mu(x)$, to be

$$\mu(x) = \max\{\delta(s) : s \text{ is a substring of } x \text{ that starts at index 1}\}.$$

Lemma 3.13. *A string x is a minimal chain element if and only if $\mu(x) \leq 1/2$.*

Proof. The proof is via two contrapositive arguments. Any string with a density greater than $1/2$ must clearly contain unmatched ones, so $\mu(x) > 1/2$ implies there is a substring s of x with an unmatched 1 which will also be an unmatched 1 in the whole string x. It follows by definition that x is not a minimal chain element in this case.

Likewise any string x that is not a minimal element must contain an unmatched 1, and, taking the substring s from index 1 up to that unmatched 1, it should be clear that the density of s must be greater than $1/2$. This follows since the last bit of s is an unmatched 1 and, if there were at least as many zeros as ones in that substring s, it would be impossible for that particular 1 to be unmatched by the matching procedure. This in turn immediately gives the max density of x is also greater than $1/2$, that is $\mu(x) > 1/2$. □

We can also say that inserting a 1 anywhere from index 2 to index z_x cannot increase $\mu(x)$ to more than $1/2$ as long as $u_x > 1$.

Theorem 3.14 (Correctness of Recursive-CTD algorithm)**.** *When Algorithm 3.12, CTD-recursive is called on $x_0 \in T_n$, it will visit the minimal element of every chain in $CTD(n)$ and every string $x \in B_n$ exactly once. More generally, when Recursive-CTD is called on vertex $x \in S_n$ it will visit every string C_y of $CTD(n)$ for which y is a descendant of x in T_n and every string x of B_n that is a descendant of x in T'_n exactly once. During the call of Recursive-CTD on any x in T_n, no more than $\lfloor n/2 \rfloor - |x| \leq \lfloor n/2 \rfloor$ recursive calls of Recursive-CTD will be stacked.*

Proof. Suppose Recursive-CTD is called with $x \in S_n$. Every $x \in S_n$ has an equal number of matched ones and zeros. If y is a child of x (by definition $x = \sigma(y)$) then since x has one fewer matched 1 than y, it must also have one fewer matched 0 than y. But the string length of x is the same as the string length of y, so there must be two more unmatched zeros in x than y. The additional unmatched zeros must appear in the first block of zeros in x. This is because the first 1 in y must occur at some index i before index z_x and changing a 1 at index i in y back to a 0 in x can't create any additional unmatched zeros in x past index i due to the matching process. This means we must have $u_x \geq u_y + 2$. Then $u_x \geq 2$ for any parent string x, since $u_x - 2 \geq u_y \geq 0$.

Thus for any string x if $u_x \leq 1$ we must conclude that x has no children. Then line 3 will correctly return and stop recursive calls for any string without children.

Now assume that $u_x > 1$ so that x has at least one child y. By the definition of σ, y has exactly one 1 at index $1 < i \leq z_x$. We cannot have $i = 1$, because then $z_y = 0$ contrary to the fact that $z_y > 0$ for all $y \in S_n$. Any value of i in the range $1 < i \leq z_x$ will correspond to a valid child string of x. To see this, suppose the string x (with $u_x > 1$) has a child y that has a 1 inserted at index i for $1 < i \leq z_x$ (index 1 is clearly not valid as mentioned previously). If the string x has any substring s (starting at index 1) such that inserting a 1 at index i would increase the density of s to more than $1/2$ that could only happen if the number of 1's in s was either equal to the number of 0's in s or one less than the number of 0's in s. But clearly this implies that s has at most one unmatched zero and the number of unmatched zeros in x (contained within the substring s) cannot exceed the number of corresponding unmatched zeros in s. This means that $u_x \leq 1$, which is a contradiction. Therefore inserting a 1 in x at index i for any $1 < i \leq z_x$ to visit child y cannot increase $\mu(y)$ to more than $1/2$ and so y is a valid child since it is a valid initial string.

In the algorithm we insert a 1 at every valid index in the range $1 < i \leq z_x$, and clearly these are all of the possible children of x as well because by the definition of σ, the preimage of x has to be $\sigma^{-1}(x) = \{y : y \text{ has a 1 inserted at index } i \text{ for } 1 < i \leq z_x\}$ so long as $u_x > 1$. So the set of "possible" children of x is clearly given by the preimage of x under sigma combined with the above argument about which of those possible strings are valid children (all of them except the string starting with a 1.)

We will now show that the correct optional arguments are passed to the recursive function call in line 4d. Clearly, for any child y, $z_y = i - 1$ so this argument is correct. To show that the other arguments are correct, namely that $u_y = m - 2$ and $U_0(y) = U_0(x) \setminus \{m, m-1\}$, we will have to consider two cases: $i > u_x$ and $i \leq u_x$. Suppose first that $i > u_x$ and so $m = \min(i, u_x) = u_x$ (this case necessarily excludes the zero string $x = x_0$ since index i must be a matched 0 in x). In y, the 1 at position i will be matched to 0 at position $i - 1$. Denote the index $u_x + 1$ by i_0 where $i_0 \leq i$. Suppose the 0 at index i_0 in x is matched to a 1 in position $j > i_0$ in x (and hence $i_0 \leq i < j$). The matching in y of positions i and $i - 1$ will change the matched pair (i_0, j) in x to a matched pair $(i_0 - 2, j)$ in y. In particular this must reduce the total number of unmatched zeros in y before index i by exactly two and thus $u_y = u_x - 2 = m - 2$. This also implies that $U_0(y) = U_0(x) \setminus \{m, m-1\}$. In the case $i \leq u_x$, the arguments are simpler. In this case, $m = \min(i, u_x) = i$. Again the 0 in position $i - 1$ is matched to the 1 in position i and all other matched 0's in y will occur after position i. Thus $u_y = i - 2 = m - 2$ and again $U_0(y) = U_0(x) \setminus \{m, m-1\}$.

Thus we have shown that Line 4 calls Recursive-CTD on every child y of x. Similarly, those recursive calls will call all the children of those children and so on, until every descendant y of x in T_n is visited. Each time such a y is visited, Recursive-

CTD also visits all the elements of C_y. So Recursive-CTD will visit all the descendants z of x in T'_n as well. In particular, this means that if Recursive-CTD is called on the all zeros string x_0, the root of T_n, it will visit every vertex of T_n and T'_n.

Each string x in T_n has no unmatched 1's since $U_1(x) = \emptyset$, and so each 1 in x is matched to a 0. Thus there can be no more than $\lfloor n/2 \rfloor$ 1's in x. Since each child in T_n contains exactly one more 1 than its parent, a call to Recursive-CTD at recursive depth ℓ that originated from an initial call to an input x is necessarily to a string x' with $|x'| = |x| + \ell \leq \lfloor n/2 \rfloor$. Thus no more than $\ell = \lfloor n/2 \rfloor - |x| \leq \lfloor n/2 \rfloor$ recursive calls of Recursive-CTD will be stacked during a call of Recursive-CTD on any particular $x \in T_n$. This proves the final claim of the theorem. □

3.6.3 Our Non-recursive Algorithm to Produce $CTD(n)$

When the Recursive-CTD algorithm is called on x_0, it will visit all the elements $x \in T_n$ and all the elements in B_n in a fixed order $\mathcal{O}_n$. In this section, we will describe our non-recursive algorithm to visit all the strings $z \in T_n$ and in B_n in the same order. Given inputs $x, y \in T_n$ with $x \leq y$ in $\mathcal{O}_n$, the algorithm will visit each $z \in T_n$ with $x \leq z \leq y$ in $\mathcal{O}_n$ and also the descendants of these z in T'_n, all in the same order as they are visited in $\mathcal{O}_n$.

We can easily use this algorithm to parallelize the process of scanning through B_n. Indeed, if we have N parallel processors, we can split up the strings in T_n into N consecutive intervals in $\mathcal{O}_n$ and have each processor simultaneously work through its own interval. We shall see more of this in Section 3.6.4. As a side benefit, the non-recursive version moves from string to string in T_n more quickly than the recursive algorithm will.

Definition 3.15. For any given string we will define a *chunk* to be any maximal run of consecutive 0's. If a string begins with a 1, we say that the first chunk is just the first maximal run of 1's preceded by an empty run of 0's. Similarly, if a string ends with a 0, the last chunk will be the last maximal run of 0's followed by an empty run of 1's.

Note that every string can be completely decomposed uniquely into chunks. Now, recall that the initial strings of CTD(n) (the minimal chain elements) are precisely those which contain no unmatched ones. We are therefore able to associate each such initial string with a matrix M with exactly 3 columns and 1 or more rows. The matrix M will essentially contain a compressed form of the information in the string pertaining to the number of unmatched zeros, matched zeros, and matched ones within each chunk of the string.

We will state this as a lemma:

Lemma 3.16. *Each initial string corresponds to a unique matrix M with 3 columns and at least 1 row. Each column corresponds to respectively, the number of unmatched*

zeros, the number of matched zeros, and the number of matched ones within each chunk respectively. The matrix row order also matches the string chunk order.

Proof. We may decompose each initial string into chunks. Because of the way matching works any unmatched zeros in such a chunk must occur at the beginning of that chunk. This is because by the matching process each matched one is always matched to the closest leftmost zero available, making it impossible for any matched zeros to proceed any unmatched zeros within any given chunk. Since each initial string does not contain any unmatched ones, each chunk can therefore be represented by a triple (a, b, c) of three numerical values. The value a represents the number of unmatched zeros at the front of the chunk and may have a value of 0. The value b represents the number of matched zeros in the chunk (following any unmatched zeros), and cannot be less than the value for c. The value c represents the number of matched ones that occur at the end of the chunk. For all but the last chunk in the string the value of c within each chunk must be at least 1, but the last chunk in the string need not contain any matched zeros or matched ones. However in every chunk there must be at least one zero (the string of all 1's is not an initial string). Therefore given any initial string it is straightforward to implement the matching process and then capture the number of unmatched zeros, matched zeros, and matched ones in each respective chunk of the string as a unique row of the matrix M. □

Note that the number of rows of M varies between 1 row and at most $\lceil n/2 \rceil$ rows depending on the string it represents (the maximal case corresponding to strings consisting entirely of 01 chunks except maybe the last chunk being just 0). Each row in the matrix M will correspond to the triple of three numerical values representing each chunk in the string.

For example, the string (with unmatched zeros underlined)

$$\underline{0}0001110010101\underline{00}$$

would be represented by the matrix

$$M = \begin{bmatrix} 1 & 3 & 3 \\ 1 & 2 & 1 \\ 0 & 1 & 2 \\ 2 & 0 & 0 \end{bmatrix}.$$

In order to facilitate visiting the chain associated with each initial string it will be useful to store some information in an additional matrix R. The matrix R will have two columns and a variable number of rows. Each row of R will contain two integers representing the starting and stopping index corresponding to each maximal substring of unmatched zeros within the initial string. The start and stop index may be the same which happens for each singular unmatched zero in the string. The

number of rows of R varies between 0 rows and at most $\lceil n/3 \rceil$ rows depending on the corresponding string (with the maximal case corresponding to strings consisting entirely of 001 chunks except maybe the last chunk being just 0 or 00). The R matrix only contains information pertaining to the unmatched zeros in the string, so in particular it is possible for the R matrix to be an empty matrix with 0 rows (as in any string of all 01 chunks for example). An empty R matrix simply indicates that the chain contains exactly one string, the initial string.

For example, the string (with unmatched zeros underlined)

$$\underline{0}0001110010101\underline{00}$$

would be associated with the matrix

$$R = \begin{bmatrix} 1 & 1 \\ 8 & 8 \\ 15 & 16 \end{bmatrix}.$$

While it is technically possible to obtain the same information contained in R from the matrix M we can typically save time in the long run by maintaining and updating both matrix M and matrix R between each initial string since such updates will always be relatively fast. In particular, we will only ever need to interact with the first two rows of either matrix M or R between any two initial strings. We will see that the updates that one needs to perform on M and R to move between strings are confined to the first and second rows of these matrices. Since these rows have either 3 or 2 elements each, this update can be done in constant time. By way of contrast, moving from string x to another string in the recursive algorithm requires deleting elements from $U_0(x)$, a list of possibly n indices. These elements can only be found by searching a list of up to n integers which still takes $\lceil \log_2(n) \rceil$ steps even if the lists are maintained in increasing order.

It should be noted that implementing this with matrices is only one of many possible methods. In particular, since we only ever modify the first two rows of both M and R the same information could be stored using linked lists or stacks which might be more memory efficient. However the process is perhaps more clearly conveyed using matrices.

Depending on the intended application of this algorithm it may or may not be necessary to keep track of the actual initial strings themselves. In some cases it will be sufficient to store and update only the matrices M and R. For example, in cases where the binary string is being used to represent some object it is possible to update the object using the information in M and R without ever needing the actual string. If the binary string were representing potential edges of a graph for example, the graph itself could be updated using only the information contained in M and R without needing the actual string. However, some applications may require the

string itself to be computed. In these cases it will be best to simply store the string and modify it directly to get each new string. While the average time to do this will typically be small, being related to the size of the first consecutive substring of 1's within the proceeding string, the worst case scenario for computation time to update the string will be $\leq c \cdot n$ for some small constant c. (Each row of M tells you what the corresponding chunk of the string is and scanning through M takes at most $n/2$ steps.) This means that if it is necessary to update each string that the worst case computation time will actually be significantly worse than not tracking the string itself at all. However it should be noted that the average computation time will be lower than the worst case computation time.

Now we may proceed to describe the non-recursive version of the algorithm. Note that in practice it is possible to simply pass the algorithm a single starting string x along with a fixed number of loops to perform (instead of an ending string y) which is computationally more efficient. However, for clarity and ease of proofs we present the algorithm in terms of two input strings, a starting string x and an ending string y.

Algorithm 3.17. *Algorithm: Non-recursive-CTD*

Input: Strings x, y in T_n with $x \leq y$ in $\mathcal{O}_n$.

Output: None. Visits each string $z \in T_n$ (and their descendants in T'_n) between x and y in the order $\mathcal{O}_n$. In particular, this algorithm can visit all of B_n.

1. Initialization.
   (a) Initialize M and R for x. Note, if $x = \hat{0}$, then $M \leftarrow [n, 0, 0]$ and $R \leftarrow [1, n]$.
   (b) Set current string s to x.
   (c) END $\leftarrow$ False
2. while $END = false$ do
   (a) call Visit-chain(s,R)
   (b) if $s = y$
   $END \leftarrow true$
   (c) else
      i. call Update-string(s,M)
      ii. call Update-R-matrix(M,R)
      iii. call Update-M-matrix(M)

3. end while

Algorithm 3.18. *Algorithm: Visit-chain*

Input: String $s \in T_n$, matrix R for x, optional Boolean function f.

Output: Visits each string in the chain corresponding to s, possibly outputs f evaluated at each element in the chain.

1. for each row i of R do
   (a) for $j \in \{R_{i,1}, R_{i,1}+1, \ldots, R_{i,2}\}$ do
      i. $s(j) \leftarrow 1$
      ii. Optionally evaluate $f(s)$.
   (b) end for
2. end for
3. for each row i of R do
   (a) for $j \in \{R_{i,1}, \ldots, R_{i,2}\}$ do
      $s(j) \leftarrow 0$
   (b) end for
4. end for

Algorithm 3.19. *Algorithm: Update-string*

Input: current string $s \in T_n$, current matrix M for s

Output: None, update s to the next string to be visited (by either changing the second 0 to a 1 or by changing the first block of 1's followed by a 0 to a block of 0's followed by a 1).

1. if $M_{1,1} > 1$ then
   $s(2) \leftarrow 1$
2. else if $M_{1,1} \leq 1$ then
   (a) for $i \in \{1, \ldots, M_{1,3}\}$ do
      $s(M_{1,1} + M_{1,2} + i) \leftarrow 0$
   (b) end for
   (c) $s(M_{1,1} + M_{1,2} + M_{1,3} + 1) \leftarrow 1$

3. end if

Algorithm 3.20. *Algorithm: Update-R-matrix*

Input: Current matrices M and R

Output: none, updates R to be the R-matrix for the updated string

1. Case 1: $M_{1,1} > 2$
   $R_{1,1} \leftarrow 3$
2. Case 2: $M_{1,1} = 2$
   Remove first row R_1 from R (note R can become an empty matrix)
3. Case 3: $M_{1,1} < 2$ and $M_{2,1} \geq 2$
   (a) Remove the first $M_{1,1}$ rows of R
   (b) $R_{1,1} \leftarrow R_{1,1} + 1$
   (c) Insert $[1, 2 \cdot (M_{1,3} - 1) + M_{1,1} + 1]$ as new first row of R
4. Case 4: $M_{1,1} < 2$ and $M_{2,1} < 2$ and $2 \cdot (M_{1,3} - 1) + M_{1,1} + M_{2,1} \neq 0$
   (a) Remove the first $(M_{1,1} + M_{2,1})$ rows of R
   (b) Insert $[1, 2 \cdot (M_{1,3} - 1) + M_{1,1} + M_{2,1}]$ as new first row of R
5. Case 5: $2 \cdot (M_{1,3} - 1) + M_{1,1} + M_{2,1} = 0$
   There is no change to R

Algorithm 3.21. *Algorithm: Update-M-matrix*

Input: Current matrix M

Output: none, updates M to be the M-matrix for the updated string

1. Case 1: $M_{1,1} > 1$
   (a) $M_{1,1} \leftarrow M_{1,1} - 2$
   (b) Insert $[0, 1, 1]$ as new first row of M
2. Case 2: $M_{1,1} \leq 1$ and $M_{2,1} > 0$
   (a) $M_{2,1} \leftarrow M_{2,1} - 1$
   (b) $M_1 \leftarrow [M_{1,1} + M_{1,2} + M_{1,3} - 1, 1, 1]$

3. Case 3: $M_{1,1} \leq 1$ and $M_{2,1} = 0$ and $M_{2,2} > 1$

   (a) $M_{2,2} \leftarrow M_{2,2} - 1$

   (b) $M_1 \leftarrow [M_{1,1} + 2 \cdot (M_{1,3} - 1), M_{1,2} - M_{1,3} + 2, 1]$

4. Case 4: $M_{1,1} \leq 1$ and $M_{2,1} = 0$ and $M_{2,2} = 1$

   (a) Replace the first two rows of M with the new first row $[M_{1,1} + 2 \cdot (M_{1,3} - 1), M_{1,2} - M_{1,3} + 2, M_{2,3} + 1]$ (this needs to be computed before the rows are deleted)

5. Case 5: $M_{1,1} \leq 1$ and $M_{2,1} = 0$ and $M_{2,2} = 0$

   This case only applies to a single initial string which is the final possible string of T_n in the order $\mathcal{O}_n$. It is therefore ruled out by the test for the end condition in the main algorithm since this is the largest possible choice for y as input to the algorithm. Note that it is possible in this algorithm for the matrix M to acquire a row of all zeros at the bottom, indicating an empty chunk, but this never effects the procedure and it is still clear which string the matrix M represents. The only string that would ever witness this row of zeros is the final possible string of T_n in the order $\mathcal{O}_n$ which is ruled out in the main algorithm as stated.

The non-recursive algorithm works by following some simple rules:

1) Always insert a 1 first if possible, always at index 2. This is always possible if the string has children, that is if $u_x > 1$. This means that if a string has a child then the algorithm always starts by visiting the first "leftmost" child and therefore is always going up and left if possible.

2) If it is not possible to insert a 1 then instead "push" the first block of ones forward (this includes "blocks" consisting of single ones). To do this means to flip every 1 in the entire first block of ones to zeros, and also simultaneously flip the single 0 bit that immediately follows that block of ones. For example, a string with $u_x \leq 1$ that starts with $0000\underline{111100}\ldots$ would be changed to $0000\underline{000010}\ldots$. Here the part that gets flipped is underlined but nothing else changes.

It is apparent that this works to visit the next unvisited ancestor's sibling. In the process of removing each successive 1 evidently a path back to the root is followed. Suppose that the block of 1's has its last 1 at some index i_1. Let a be the ancestor of x that has all 0's up to the index i_1 and has a 1 at index i_1. If the first block of 1's has k ones then this ancestor would be $a = \sigma^{k-1}(x)$. Technically, for a string with a single 1, where $k = 1$, this would be $\sigma^0(x)$. We can define $\sigma^0(x)$ as the identity function, so the ancestor in question would be x itself, i.e. if $k = 1$ then $a = x$. The farthest rightmost child in the subtree of a is the string with ones inserted at the rightmost

possible indices (in front of index i_1 by definition) each time, until no further ones can be inserted. So the rightmost descendant of a is a string with all zeros followed by all ones until index i_1 is reached, and such that this descendant x of a has $u_x \leq 1$. If u_x was greater than 1, then x would have children and therefore would not be the rightmost possible descendant of a. So this shows that if $u_x \leq 1$ then x must indeed be the rightmost descendant of its ancestor $a = \sigma^{k-1}(x)$. Therefore the next node to be visited in the tree has to be the sibling of the ancestor a (since we are following a depth search first pattern). But the sibling of a is the string with the first 1 of a (at index i_1) shifted to the right one index (to index $i_1 + 1$). There must be a zero after this 1 because of the definition of a. The only possible string without a zero available after index i_1 has to be the final string in the entire tree, consisting of all zeros followed by all ones and with $u_x \leq 1$, so the algorithm also knows it must terminate in that case.

Theorem 3.22 (Correctness of Non-recursive CTD algorithm)**.** *When called on inputs x and y in T_n with $x \leq y$ in $\mathcal{O}_n$ Algorithm 3.17, Non-recursive-CTD, visits every element $z \in T_n$ in the interval $[x, y]$ in $\mathcal{O}_n$ and all elements in the chains C_z in $CTD(n)$, all in the order $\mathcal{O}_n$. It will visit each of these strings exactly once.*

Proof. Recall that $S_n = \{x \in B_n : U_1(x) = \emptyset\}$ is the set of minimal or initial elements of the chains C_x in $CTD(n)$. We define a function $Rev : S_n \to S$, where $Rev(x)$ is the reversed binary string. For example $Rev(01001) = 10010$. Next we define a function $Val : B_n \to \mathbb{N}$ which returns the binary value of a given string in B_n. Finally we define the function $f : S_n \to \mathbb{N}$ by $f(x) = Val(Rev(x))$ for all $x \in S_n$. For example $f(01001) = Val(Rev(01001)) = Val(10010) = 18$.

We first describe how the algorithm works when x is the first string in $\mathcal{O}_n$ and y is the last.

Let S_{first} and S_{last} represent the first and last initial strings visited by the Recursive-SCD algorithm, that is S_{first} is the initial string comprised of all zeros while S_{last} is the initial string comprised of $\lceil n/2 \rceil$ consecutive 0's followed by $\lfloor n/2 \rfloor$ consecutive 1's. (Note that $f(S_{first}) = 0$ and that $f(S_{last}) = 2^n - 2^{\lceil n/2 \rceil}$.) We define a function $Next : S_n \setminus S_{last} \to S_n$ by $Next(x)$ is the next initial string visited by the non-recursive algorithm. Since the algorithm clearly defines a unique next initial string for every string this is a well-defined function. We will also define a function $Previous : S_n \setminus S_{first} \to S_n$ which will map each initial string to the preceding string in the non-recursive algorithm. We give an explicit high level description of $Next$ and $Previous$ as follows.

Each initial string x may begin with a number of unmatched zeros u_x. If $u_x > 1$ then the $Next$ function will simply change the second 0 in the string to a 1. Otherwise the $Next$ function will change the first block of 1's followed by a 0 in the string to a block of 0's followed by a 1. This process uniquely defines the next string in the algorithm for all strings except the last initial string S_{last}.

The *Previous* function will be the inverse of the *Next* function. Therefore, for any initial string with a 1 in the second position the *Previous* function will change that to a 0 in the second position. Otherwise, the *Previous* function will change the first 1 and a block of 0's before it into a 0 with a block of 1's before it. The size of the block of 0's changed must be such that no larger block of 0's could be changed without creating unmatched 1's which are not allowed in initial strings. Such a change would necessarily leave exactly zero or one unmatched 0's at the beginning of the string. Converting any fewer 0's to 1's would leave at least two unmatched 0's at the beginning and thus *Previous* would fail to be an inverse to *Next*. This process uniquely defines the previous initial string in the algorithm for all strings except the first initial string S_{first} since it is the only string not containing any 1's. Thus the *Previous* function is also uniquely defined and is necessarily an inverse of *Next* by definition. That is, we must have $Next(Previous(x)) = x$ and $Previous(Next(x)) = x$ for all $x \in S_n \setminus (S_{first} \cup S_{last})$.

We may now show that for all $x \in S_n \setminus (S_{first} \cup S_{last})$ that $f(Previous(x)) < f(x) < f(Next(x))$. Let x be an arbitrary string in S_n. First to show $f(x) < f(Next(x))$ we consider two cases. If x begins with at least two unmatched zeros, then $Rev(x)$ ends with two zeros, and $Next(x)$ inserts a 1 into the second position so that $Rev(Next(x))$ is the same as $Rev(x)$ except that $Rev(Next(x))$ ends in a 10. From this it is clear that $f(x) < f(Next(x))$. To be precise, in this case $f(x) + 2 = f(Next(x))$. In the other case we know by definition that $Next(x)$ changes the first block of 1's followed by a 0 to 0's followed by a 1. Let k be the index of the last 1 in the string $Rev(x)$. From the algorithm definition it is clear that finding $Rev(Next(x))$ is equivalent to binary addition of the string with a single 1 in index k to $Rev(x)$. Thus it is clear in this case also that $f(x) < f(Next(x))$. More precisely, in this case we have $f(x) + 2^{k-1} = f(Next(x))$. Since x was arbitrary and by definition $Next(Previous(x)) = x$ then this also shows that $f(Previous(x)) < f(x)$. Thus as the algorithm proceeds through every initial string the value of $f(x)$ for each string is strictly increasing.

Now suppose for contradiction that there exists at least one string which is not visited by the algorithm. Let s be the string that is not visited by the algorithm with the smallest value for $f(s)$. Then necessarily $Previous(s)$ must be visited by the algorithm since $f(Previous(s)) < f(s)$, otherwise s would not have been the smallest valued missed string. But since the algorithm visits $Previous(s)$ and $Next(Previous(s)) = s$ then the algorithm must visit the string s as well. This contradiction shows that there cannot be any initial strings that are missed by the algorithm. It also follows easily from the fact the f is a strictly increasing function as each string is visited that no string can be visited more than once. Thus every initial string must be visited exactly once by the algorithm as claimed.

The algorithm will terminate precisely when there is at most one unmatched 0 at the beginning of the string and there are no 0's at all after the first block. This can only happen for initial strings with exactly one block of 0's followed by 1's. However for all initial string with exactly one block of 0's and 1's, all of them have at least two unmatched 0's at the front except for the block with exactly $\lceil n/2 \rceil$ consecutive 0's followed by $\lfloor n/2 \rfloor$ consecutive 1's. We refer the this string as S_{last}. Any fewer number of 1's in the string would have two or more unmatched 0's at the beginning of the string, while any more 1's in the string would necessarily have unmatched 1's since the total number of 1's would exceed the total number of 0's. It follows therefore that there is a single unique string that satisfies the end condition. Furthermore it is clear that for this string neither one of the two cases for the $Next$ function are applicable, and this is the only such sting with neither condition applicable. So the algorithm must terminate upon encountering the string S_{last}. Additionally the string S_{last} must be visited by the algorithm by the argument above showing that no strings are missed. Therefore the algorithm must terminate, as desired.

Clearly, the way the algorithm is defined, it will also visit the elements of the chains of $CTD(n)$ in the order $\mathcal{O}_n$. It will behave the same way on intervals $[x, y]$ in $\mathcal{O}(n)$ as it is defined to start the scanning at x and stop once it gets to y and its chain. □

3.6.4 Our Parallel Algorithm to Produce $CTD(n)$

Given Algorithm 3.17, Non-recursive CTD, is it clear how to parallelize this algorithm to distribute the job of visiting every string in T_n and B_n amongst an arbitrary number of parallel processors.

Algorithm 3.23. *Algorithm: Parallel-CTD*

Input: Integer $N \geq 1$ and methods to assign tasks to N parallel processors.

Output: None. Visits every string in T_n and in B_n in the order $\mathcal{O}_n$

1. Initialize. Find $x_i, y_i \in T_n$ for $1 \leq i \leq n$ such that $x_1 \leq y_1 \leq x_2 \leq y_2 \leq \cdots \leq x_N \leq y_N$ in $\mathcal{O}_n$, the intervals $[x_i, y_i]$ in $\mathcal{O}_n$ partition T_n, and each interval $[x_i, y_i]$, $1 \leq i \leq N$ contains (plus or minus 1) $1/N$ fraction of all strings in T_n.

2. For $1 \leq i \leq n$, call Non-recursive-CTD(x_i, y_i) on processor i.

In this thesis we will only use this algorithm as a subroutine to answer Problems 3.1-3.3 on monotone Boolean functions $f : B_n \to \{0, 1\}$. Since we are only interested in the number of computations of f that are made, the initialization step may be

carried out in any way as it involves no computation of f. The most obvious way is to run Non-recursive-CTD on the whole of B_n. The elements $z_1, \ldots, z_m$ of T_n will be produced one by one in the order of $\mathcal{O}_n$ where $m = |T_n| = M_n = \binom{n}{\lfloor n/2 \rfloor}$. We can select the x_i and y_i from the $z_j \in T_n$ as they are produced. Set $j_i = \lfloor (i/N)M_n \rfloor$ for $0 \leq i \leq N$. For $1 \leq i \leq N$, select $x_i = z_{j_{i-1}+1}$ and $y_i = z_{j_i}$. The intervals will partition T_n and $[x_i, y_i]$ will contain $j_i - j_{i-1} = \lfloor (i/N)M_n \rfloor - \lfloor ((i-1)/N)M_n \rfloor$ elements which is indeed within 1 of $(1/N)|T_n| = (1/N)M_n$.

A better way to initialize the parallelization is by using an algorithm to determine the kth minimal element in the order $\mathcal{O}_n$ instead of running Non-recursive-CTD on all of B_n. This could be used to quickly determine the x_i and y_i needed for each interval in the parallelization. We provide such an algorithm below but omit the proof of correctness. This algorithm ends after at most n loops and therefore admits a much faster method of initializing the parallelization.

Algorithm 3.24. *Algorithm: kth element*

Input: string length n, desired minimal string k with $1 \leq k \leq |T_n| = \binom{n}{\lfloor n/2 \rfloor}$

Output: kth minimal element in the tree order of T_n

1. $x \leftarrow 0_n$ (initialize x)
2. $z_x \leftarrow n$ (initial value for z_x)
3. $u_x \leftarrow n$ (initial value for u_x)
4. $K \leftarrow \binom{n}{\lfloor n/2 \rfloor} + 1 - k$
5. for $i \in \{0, \ldots, n-1\}$ do
   (a) if $u_x < 2$ or $(n-i) < 2$ then
   return x
   (b) end if
   (c) $y \leftarrow x$
   (d) $y_{n-i} \leftarrow 1$ (change the 0 to a 1 at index $n-i$ in y)
   (e) $z_y \leftarrow n - i - 1$ (compute new values for y)
   (f) $u_y \leftarrow \min(n-i-2, u_x - 2)$
   (g) $T \leftarrow \binom{z_y}{\lfloor u_y/2 \rfloor}$
   (h) if $K < T$ then
   i. $x \leftarrow y$ (in this case keep the updates to y)

ii. $z_x \leftarrow z_y$

iii. $u_x \leftarrow u_y$

(i) else if $K > T$ then

$K \leftarrow K - T$ (in this case do not keep the updates to y)

(j) else if $K = T$ then

return y

(k) end if

6. end for

3.7 Algorithms for the Main Problems

3.7.1 Our Algorithms

Suppose we are given a monotone Boolean function $f : B_n \to \{0, 1\}$ and wish to solve Problem 3.1, list the elements of $I(f) = \{x \in B_n : f(x) = 0\}$, Problem 3.2, list the maximal elements of $I(f)$, or Problem 3.3, calculate $|I(f)|$. We will use our algorithms to produce all the chains C_x in $CTD(n)$ and their minimal elements $x \in T_n$. Since these C_x's are chains and f is monotone we can use binary search to find the thresholds at which $f(y)$ is last 0 as y increases through C_x.

Recall that if $x \in T_n$ with $k = |U_0(x)|$ (the number of unmatched 0's in x) then $C_x = \{y_0, \ldots, y_k\}$ has $k + 1$ elements where $y_i = \tau^i(x)$ is x with its i left-most unmatched 0's flipped to 1's. In particular, we have $y_0 \lessdot y_1 \lessdot \cdots \lessdot y_k$ in B_n. Since f is monotone we have $f(y_0) \leq f(y_1) \leq \cdots \leq f(y_k)$. We define the *threshold* of f with respect to C_x to be $t(f, x) = \max\{i : f(y_i) = 0\}$ with the special value $t(f, x) = -1$ if every f value is 1. Thus $f(y_i) = 0$ if $i \leq t(f, x)$ and $f(y_i) = 1$ if $i > t(f, x)$.

Lemma 3.25. *If $f : B_n \to \{0, 1\}$ is monotone and C_x is a chain in B_n in $CTD(n)$, then $t(f, x)$ can be found with at most $\lceil \log_2(|C_x| + 1) \rceil \leq \lceil \log_2(n + 2) \rceil$ evaluations of f on elements of C_x.*

Proof. If the number of elements in C_x is $N = 2^m - 1$ then $t(f, x)$ can be found by binary search with only m evaluations of f. Indeed, if $m = 1$ then $C_x = \{x\}$ and evaluating $f(x)$ is enough. If $f(x) = 0$ then $t = 0$ and if $f(x) = 1$, then $t = -1$. We will complete the proof by induction. Suppose now that for some $m \geq 1$, we have proven that m evaluations are enough. We will show that $m + 1$ evaluations are enough for a chain C_x of $2^{m+1} - 1$ elements as follows. Evaluate $f(y)$, where y is the middle element of the chain. If $f(y) = 0$ then we are reduced to finding the threshold amongst the $2^m - 1$ elements of the chain that come after y. If $f(y) = 1$, we are reduced to finding the threshold amongst the $2^m - 1$ elements of the chain that

come before y. By the inductive hypothesis, both of these cases can be handled by m further function evaluations for a total of $m+1$ evaluations.

Suppose now that the number of elements in the chain, N, satisfies $2^{m-1} \leq N \leq 2^m - 1$. We can then pad the sequence of f values with $2^m - 1 - N$, 1's at the end to bring it to a list of length $2^m - 1$ and then find the threshold in that list of values using m tests of the values within that sequence and so at most m evaluations of f. The threshold in that list will be the same as $t(f, x)$. Note that we do not actually have to double the amount of memory used for the padding. We can instead extend the definition of f to just return 1 if called for a position beyond the length N of the chain.

Now we prove the bound on the number of evaluations. Since $2^{m-1} < N + 1 \leq 2^m$ we have that the number of evaluations needed is $m = \lceil \log_2(N+1) \rceil$. The maximum possible number of elements in a chain C_x in $CTD(n)$ or in any chain in B_n is $n+1$. A chain can contain at most one element y at each possible rank $0 \leq r(y) \leq n$. Thus $m \leq \lceil \log_2(n+2) \rceil$ as claimed.

□

With this lemma in hand it is easy to design our algorithms for Problem 3.1-3.3. Since the algorithms for these problems have so much in common, we will give all them combined into one main algorithm. Any of our algorithms for producing $CTD(n)$ can be used as the main subroutine, but since we are primarily interested in parallelization, we will use Algorithm 3.23, Parallel-CTD for this purpose. Let $M_n = \binom{n}{\lfloor n/2 \rfloor}$ be the number of chains in $CTD(n)$.

Algorithm 3.26. *Algorithm: Main (combined algorithm for Problems 3.1-3.3)*

Input: A monotone Boolean function $f : B_n \to \{0,1\}$, $p \in \{3.1, 3.2, 3.3\}$, methods to send and receive messages to and from N parallel processors.

Output: The solution to Problem p for f. If $p = 3.1$, all the elements of $I(f)$ are listed. If $p = 3.2$, all the maximal elements of $I(f)$ are listed. If $p = 3.3$, $|I(f)|$ is returned.

1. Initialization.
   (a) If $p = 3.1$, do nothing.
   (b) If $p = 3.2$, each processor i sets $M_i \leftarrow \emptyset$ to be the current list of potential maximal elements of $I(f)$ that it has found.
   (c) If $p = 3.3$, each processor sets $C_i \leftarrow 0$ to be the current count of elements in $I(f)$ that it has found.
2. Scanning B_n.

(a) Distribute $\lfloor (1/N)M_n \rfloor \pm 1$ chains in $CTD(n)$ to each processor using Algorithm 3.23, Parallel-CTD.

(b) When a processor i is scanning $C_x = \{y_0, \ldots, y_k\}$ it first finds threshold $t_i(f, x)$ using the binary search algorithm laid out in Lemma 3.25.

i. If $p = 3.1$, processor i does nothing if $t_i = -1$ and otherwise sends elements y_0 through y_{t_i} of its chain to the central processor to list out.

ii. If $p = 3.2$, processor i does nothing if $t_i = -1$ and otherwise adds y_{t_i} to M_i.

iii. If $p = 3.3$, processor i does nothing if $t_i = -1$ and otherwise adds $t_i + 1$ to C_i.

3. Finishing up.

(a) If $p = 3.1$, nothing more is done.

(b) If $p = 3.2$, all processors i send M_i to the central processor. Then the central processor does the following.

i. $M \leftarrow \bigcup_i M_i$

ii. $M' \leftarrow \emptyset$

iii. While $M \neq \emptyset$ remove first element y of M if there is $z \in M$ with $y < z$ and otherwise add y to M' and remove all elements z of M with $z < y$.

iv. Return M'.

(c) If $p = 3.3$, all processors i send C_i to the central processor. Then the central processor returns $C = \sum_i C_i$.

Theorem 3.27. *Algorithm 3.26 gives the correct outputs.*

Proof. Suppose $p = 3.1$. Then in Line 2bi, processors i's current chain $C_x = \{y_0, \ldots, y_k\}$ has only elements $y_0, \ldots, y_{t_i}$ with f value 0, i.e. in $I(f)$. Note that there are no such elements when $t_i = -1$. These elements are sent back one by one to the central processor to list out. All the processors together go through all of the chains in $CTD(n)$. These chains partition B_n so all elements of $I(f)$ are listed by the central processor.

Suppose now that $p = 3.2$. Then in Line 2bii, C_x has no elements in $I(f)$ if $t_i = -1$. If $t_i \geq 0$, then y_{t_i} is the only possible element of the current chain C_x which could be a maximal element of $I(f)$. M_i will wind up being these maximal elements in all the chains that processor i considers. In Line 3bi, $\bigcup_i M_i$ is the set of all the elements of $I(f)$ that are maximal with respect to the chain of $CTD(n)$ that they are in. The maximal elements of $I(f)$ are necessarily the maximal elements of $I(f)$ in their own chains and so must all be contained in M. Line 3biii winnows down this set M to the

true set M' of maximal elements of $I(f)$. Each element y in M is considered exactly once. If y is not a maximal element of $I(f)$ then there is another maximal element $z \in I(f)$ with $y < z$. This z will be the maximal element of its chain and so will initially wind up in M. If z is considered before y, y will not be added to M'. If z is considered after y, then again y will not be added to M'. If y is a maximal element, then it will be added to M'. Thus M' will end up being the set of maximal elements of $I(f)$. Note that it can be possible to filter down elements of M dynamically as elements are added to it which would be computationally more efficient, but this involves a more complex algorithm.

Suppose finally that $p = 3.3$. Then in Line 2biii, exactly $t_i + 1$ elements of the current chain belong to $I(f)$. C_i will wind up being the number of elements of $I(f)$ in the chains that processor i considers. In Line 3c, C will wind up being the number of elements in $I(f)$ in all of the chains of $CTD(n)$ and hence in all of B_n. So $C = |I(f)|$. □

3.7.2 Hansel's Algorithm

Hansel's algorithm [Han66] is the classical algorithm for finding the thresholds $t(f, x)$ for each chain C_x in $CTD(n)$. We introduce it here in order to compare its performance to the performance of our algorithms for this problem. This will be done in Section 3.8

We will quote Knuth's [Knu11] presentation of the algorithm. Recall that we have been using the presentation of $CTD(n)$ as given in [GK76]. Hansel's algorithm uses the initial presentation of $CTD(n)$ as first given in [dBvETK51]. There $CTD(n)$ is given as the Christmas Tree Pattern, or $CTP(n)$. This is an arrangement of the chains in $CTD(n)$ in rows with the elements of the chains read in increasing order from left to right. The rows are centered so that strings x with $|x| = k$ are in column k of the table. We quote the exposition of $CTP(n)$ given in [Knu11]. $CTP(1)$ consists of a singe row with two columns containing the bitstrings $0 \in B_1$ and $1 \in B_1$. $CTP(2)$ consists of two rows, given by

$$10$$

$$00\ 01\ 11$$

In general, $CTP(n+1)$ is generated recursively from $CTP(n)$ by replacing each row $\sigma_1\sigma_2\ldots\sigma_s$ by the two rows

$$\sigma_2 0 \ldots \sigma_s 0$$

$$\sigma_1 0\ \sigma_1 1 \ldots \sigma_{s-1} 1\ \sigma_s 1$$

(If $s = 1$ the first of these two rows is omitted.)

It is shown in [GK76] that the rows of $CTP(n)$ are the chains in $CTD(n)$.

Following Knuth's notation, given $\sigma \in B_n$ we have the function $r(\sigma)$ which returns the row number of σ in the $CTP(n)$. We also have the function $s(m)$ which returns the number of strings in row m of the $CTP(n)$ for $1 \leq m \leq M_n$. This is the number of elements in the corresponding chain. The function $s(m)$ is uniquely determined by the specific row ordering found in the $CTP(n)$. The algorithm also uses the nimsum, $\oplus$, operation on B_n. The nimsum of $x_1, \ldots, x_k \in B_n$ is $z = x_1 \oplus x_2 \oplus \cdots \oplus x_k \in B_n$ where $z_i = \sum_{j=1}(x_j)_i \mod 2$ for $1 \leq i \leq n$. Finally, we also need the function $\chi(m, k)$ which returns the bit string in column k of row m of the CTP, where $(n + 1 - s(m))/2 \leq k \leq (n - 1 + s(m))/2$ (since each row is centered around one or two middle columns in the $CTP(n)$.

Hansel's algorithm will determine a sequence of threshold values $t(1), t(2), \ldots, t(M_n)$ such that

$$f(\sigma) = 1 \iff |\sigma| \geq t(r(\sigma)).$$

using at most two evaluations of f for each of the M_n chains in $CTP(n)$. Note that $t(i)$ is the threshold of the minimum *rank* of an element in the ith chain whose f value is 1. In contrast, our t value is the maximum location *within the chain* of an element with an f value of 0. Each of these thresholds is easily obtained from the other as the minimum 1 always comes one step after the maximum 0.

Hansel's algorithm is now as follows.

Algorithm 3.28. *Hansel's Algorithm.*

Input: Monotone $f : B_n \to \{0, 1\}$.

Output: Threshold values $t(i)$ each C_i in $CTP(n)$.

1. for $m \in \{1, \ldots, M_n\}$ do
   (a) $a \leftarrow \frac{n+1-s(m)}{2}$
   (b) $z \leftarrow \frac{n-1+s(m)}{2}$
   (c) while $z > a + 1$ do (perform a binary search using already computed threshold values)
      i. $k \leftarrow \left\lfloor \frac{a+z}{2} \right\rfloor$
      ii. $\sigma \leftarrow \chi(m, k-1) \oplus \chi(m, k) \oplus \chi(m, k+1)$
      iii. if $k \geq t(r(\sigma))$ then
         $z \leftarrow k$
      iv. else if $k < t(r(\sigma))$ then
         $a \leftarrow k$
   (d) end while

(e) if $f(\chi(m,a)) = 1$ then
$t(m) \leftarrow a$

(f) else if $a = z$ then
$t(m) \leftarrow a + 1$

(g) else
$t(m) \leftarrow z + 1 - f(\chi(m,z))$

2. end for

For justification that Hansel's Algorithm is correct, see [Knu11].

Note that Hansel's Algorithm can be used as a subroutine to solve Problems 3.1-3.3. Once you have the threshold $t(i)$ for a chain in $CTP(n)$, i.e. also a chain $CTD(n)$, you have the thresholds for $I(f)$ restricted to that chain. Thus you can use Hansel's Algorithm to iterate through the chains instead of the Parallel-CTD algorithm.

3.8 Performance of Our Algorithms Versus Hansel's Algorithm

We will define the *space complexity*, $s(A,n)$, of our algorithms A to be the number of memory locations used by A as a function of n. We will define the *time complexity*, $t(A,n)$, as the number of computations of the input Boolean function that are made as a function of n.

We will only give upper and lower bounds on these space and time complexities in terms of simpler functions $k(n)$. Often we will give bounds that are only true up to a constant factor. For instance, we will write statements such as $t(A,n) = \Omega(k(n))$ to mean $t(A,n) \geq ck(n)$ where $c > 0$ is some fixed but not precisely determined constant. We will also write statements such as $s(A,n) = O(k(n))$ to mean $s(A,n) \leq Ck(n)$ where $C > 0$ again is some fixed but not precisely determined constant. We write $f(n) = \Theta(f(n))$ to mean that we have both $f(n) = O(k(n))$ and $f(n) = \Omega(k(n))$. The constant factors are highly dependent on the internal workings of the computer languages used to write the algorithms and the computers that they are run on, so there is no value in trying to determine them precisely for any one choice of language or computer.

If you are given an arbitrary Boolean function f, in particular one that is not monotone, then there can be no non-trivial upper bound on the number of computations of f you might need to make in order solve Problems 3.1 or 3.3.

Theorem 3.29. *Given a general Boolean function $f : B_n \to \{0,1\}$, any algorithm A to solve Problem 3.1 or 3.3 must, in the worst case, evaluate $f(x)$ for every $x \in B_n$*

or be able to break RSA cryptography. This remains true even if the algorithm to compute f is accessible to A. Thus for any algorithm A that works on arbitrary Boolean functions f, we have that $t(A, n) = 2^n$.

Proof. We define two Boolean functions f and g on B_n. We will have $f(x) = 1$ for all $x \in B_n$, except $f(x_0) = 0$ for a single $x_0 \in B_n$ which may be chosen to be an arbitrary element of B_n. On the other hand, we will have $g(x) = 1$ for all $x \in B_n$. Thus $I(f) = \{x_0\}$ and $I(g) = \emptyset$.

The algorithms for f and g will be designed using RSA cryptography [RSA78]. Let $RSA(x)$ the RSA encoding of bitstring $x \in B_n$ defined in terms of the public key and such that the private key is known only to the algorithm designer. The designer picks $x_0 \in B_n$ however they like and designs the algorithm for f be $f(x) = 1$ if $RSA(x) \neq RSA(x_0)$ and $f(x) = 0$ if $RSA(x) = RSA(x_0)$. The designer now picks a string w such that $RSA(x) \neq w$ for all $x \in B_n$ and designs the algorithm for g to be $g(x) = 1$ if $RSA(x) \neq w$ and $g(x) = 0$ if $RSA(x) = w$. Thus we have $I(f) = \{x_0\}$ and $I(g) = \emptyset$ as was desired.

Suppose we have an algorithm A for Problem 3.1, list $I(F)$, or for Problem 3.3, return $|I(F)|$. Suppose A is given access to the Boolean function F or even access to the code for F. In order for A to distinguish between $F = f$ and $F = g$ it must, in the worst case, evaluate $F(x)$ for every $x \in B_n$. Even if the algorithm A has access to the code for f or g, it will have access to only $RSA(x_0)$ or w. If A is somehow able to not evaluate f or g on every input, it will have to somehow be able to get information about the RSA decodings of these strings. There are no publicly known techniques for doing this. □

Knuth notes that one can do only a little better for monotone Boolean functions.

Theorem 3.30. *Given a monotone Boolean function $f : B_n \to \{0, 1\}$ any algorithm A must, in the worst case, make $M_n = \binom{n}{\lfloor n/2 \rfloor}$ evaluations of f or be able to break RSA cryptography. This remains true even if A has access to the code for computing f. Thus for any algorithm A that works on arbitrary Boolean functions f, we have that $t(A, n) = \Omega(2^n/\sqrt{n})$ or that A is able to break RSA cryptography.*

Proof. We define two monotone Boolean functions f and g on B_n. We define $f(x) = 1$ if $|x| \geq \lceil n/2 \rceil$ and $f(x) = 0$ otherwise. We define $g(x) = f(x)$ for all x except we define $g(x_0) = 1$ for exactly one x_0 with $|x_0| = \lfloor n/2 \rfloor$. This x_0 may be chosen arbitrarily. Thus the set of maximal elements $M(f)$ of $I(f)$ is $M(f) = \{x \in B_n : |x| = \lfloor n/2 \rfloor\}$ and $M(g) = M(f) \setminus \{x_0\}$.

The designer of the code for f and g uses RSA cryptography. Let $RSA(F) = \{RSA(x) : x \in I(F)\}$ be the RSA encodings of all bitstrings $x \in I(f)$. For $F = f$ or $F = g$ set $F(x) = 0$ if $RSA(x) \in RSA(F)$ and $F(x) = 1$ if $RSA(x) \notin RSA(F)$. Thus f and g will have the claimed properties.

Suppose we have an algorithm A to solve Problems 3.1-3.3 for monotone Boolean functions F. In order for this algorithm to distinguish between inputs $F = f$ and $F = g$ it will, in the worst case, have to evaluate $F(x)$ for all M_n inputs x with $x = \lfloor n/2 \rfloor$. This remains true even if A has access to the code for F. If $F = f$ or $F = g$, all A will have access to is $RSA(F)$, a set of RSA encodings. If A could make fewer than M_n evaluations of F, it will have to somehow be able to get information on the RSA decodings of these strings. There are no publicly known techniques for doing this. Note that A can't just count the number of strings in the lookup tables that f and g are using as there is no guarantee that those strings are the RSA encoding of any x.

For the lower bound on $t(A, n)$, we need only use the standard fact that Stirling's asymptotic formula $n! \sim (n/e)^n\sqrt{2\pi n}$ as $n \to \infty$ implies $M_n \sim \sqrt{\frac{2}{\pi}}(2^n/\sqrt{n})$ as $n \to \infty$.

□

It should be noted that [Knu11] only gives this $t(A, n) = M_n$ lower bound for the number of function evaluations in the case that you only have access to the *results* of the computations. Our result adds that even having access to the code for f is unlikely to improve this lower bound.

The following theorem gives bounds on the space complexities of our algorithms for Problems 3.1-3.3 and shows how this compares with the space complexity of Hansel's algorithm. We see that the space complexity of our algorithms is polynomial in n while Hansel's Algorithm has space complexity that is exponential in n.

Theorem 3.31. . *We have the following space complexity bounds.*

1. *The space complexity of Algorithm 3.12, Recursive-CTD, is $s(A, n) = O(n^2)$.*
2. *The space complexities of Algorithm 3.17, Non-recursive-CTD, Algorithm 3.23, Parallel-CTD, and Algorithm 3.26, Main Algorithm, for Problems 3.1 and 3.3 are all $s(A, n) = O(n)$ per processor.*
3. *The space complexity for Algorithm 3.26, Main Algorithm, for Problem 3.2 is $s(A, n) = O(M_n) = O(2^n/\sqrt{n})$.*
4. *The space complexity for Algorithm 3.28, Hansel's Algorithm, for Problems 3.1-3.3 is $s(A, n) = \Omega(2^n/\sqrt{n})$*

Proof. It is clear that Recursive-CTD must maintain z_x, u_x, and $U_0(x)$ for all x for which there are calls of Recursive-CTD stacked. Note z_x and u_x are single indices and $U_0(x)$ is a list of up to n indices. By Theorem 3.14 at most $n/2$ function calls are stacked at any one time. This gives the $O(n^2)$ upper bound in Statement 1.

As the definition of the M and R matrices in Subsection 3.6.3 make clear, these matrices have at most $n/2$ rows and a constant number of indices per row. Since maintaining M and R is the most space intensive task Non-Recursive-CTD and Parallel-CTD face, this gives the $O(n)$ bound. The Main algorithm uses Parallel-CTD and, if working on Problems 3.1 or 3.3, need not maintain more memory. This gives the $O(n)$ upper bound here as well.

We are only able to prove the weaker $O(2^n/\sqrt{n})$ bound on the Main Algorithm for Problem 3.2. Its most expensive space requirement is to maintain the lists M_i and M of all the potential maximal elements of $I(f)$. As the function defined by $f(x) = 1$ if and only if $|x| > n/2$ shows, the set of maximal elements that is contained in these might contain M_n elements (and at most M_n) elements. Since $M_n = \Omega(2^n/\sqrt{n})$, this gives the bound in Statement 3.

A careful analysis of Line 1 in Hansel's Algorithm, one that is carried out in [Knu11], shows that the calculation of a threshold $t(i)$ is carried out in terms of thresholds $t(j)$ for earlier rows $j < i$ in $CTP(n)$. Thus the list of t values for all $M_n = \Omega(2^n/\sqrt{n})$ rows of $CTP(n)$ must be maintained. □

The following theorem gives bounds on the time complexities of our Main Algorithm for Problems 3.1-3.3 and shows how this compares to the time complexity of Hansel's algorithm.

Theorem 3.32. *We have the following statements.*

1. *The time complexity of Algorithm 3.26, the Main Algorithm running on N processors is $t(A, n) = O((1/N)(2^n \log(n)/\sqrt{n})$ per processor.*

2. *The time complexity of Algorithm 3.28, Hansel's Algorithm is $\theta(2^n/\sqrt{n})$.*

Proof. We first prove Statement 1. We see that each processor gets $(1/N)M_n = O((1/N)2^n/\sqrt{n})$ of the chains of $CTD(n)$. By Lemma 3.25, each processor need only make $O(\log(n))$ evaluations of f per chain. Since the processors operate in parallel, this gives the bound in the statement.

We now prove Statement 2. As mentioned in the proof of Theorem 3.31, each computation of a threshold in Hansel's Algorithm may require any of the previously computed of thresholds. Thus there is no way to divide the chains up among processors that can work in parallel. As proved in [Knu11], Hansel's algorithm makes only 2 computations of f per chain. Thus its time complexity is $t(A, n) = \Theta(M_n)$ which gives the bound. □

As mentioned in the introduction of this chapter, Theorems 3.31 and 3.32 tell us that our Main Algorithm has an important advantage over Hansel's algorithm. The time complexity of the Main Algorithm is a factor $(1/N)\log(n)$ less than that of Hansel's Algorithm. If your number N of parallel processors is within the thousands this is

significant. Furthermore the space complexity of the Main Algorithm is $O(Nn)$ (for N processors) as opposed to the $\Omega(2^n/\sqrt{n})$ of Hansel's Algorithm. This significantly extends the limits of computational exploration of conjectures on monotone Boolean functions such as Conjecture 2.3. We say more about that in the next and final section of this chapter.

3.9 Testing of the Bunk Bed Conjecture

We first came at these Problems through a special case of Problem 3.6 that originated in computationally testing Conjecture 2.3 and several variants of it. Let $B = BB(G,T) = (V,E)$ be a bunk bed graph. Let x and y be vertices of B. Define the Boolean function $f_{xy} : \mathcal{P}(E) \to \{0,1\}$, such that for all subsets F of the edge set E we have $f_{xy}(F) = 1$ if and only if vertices x and y are connected in (V,F). It is easy to see that f_{xy} is monotone. This is because if a connection event holds true in (V,F) it must hold true in (V,F') for all $F' \supseteq F$; adding more edges cannot destroy connectivity.

Conjecture 2.3 is thus about the number of subgraphs (V,F) of B for which $f_{xy}(F) = 1$, i.e. about the size of the filter $|F(x,y)| = |f_{xy}^{-1}(1)|$. Conjecture 2.3 asserts $|F(x_0,y_0)| \geq |F(x_0,y_1)|$ for all x, y. As noted, we used our algorithms to test various cases of Conjecture 2.3.

In practice we could greatly improve the efficiency of our counting of $|F(x,y)|$ in two ways. If while searching the vertices F of the associated tree T_n' on $B_n = \mathcal{P}(E)$ we discovered that $f_{xy}(F) = 1$, then we could add y to the count for $|F(x,y)|$ for all $F' \geq F$. This was particularly helpful when $F \in T_n$ as the entire subtree of T_n rooted at F no longer had to be tested.

The second way we could improve performance was that as the algorithm scanned T_n' it mostly moved from sets F to $F \cup \{e\}$. We would maintain the list of connected components of the current spanning subgraph (V,F) as we went. Updating these components after the addition of an edge was a trivial task. Either the edge joined two components or it landed entirely within one component. Testing $f_{xy}(F)$ was then easy too. Either x and y were both in the same component and $f_{xy}(F) = 1$ or otherwise $f_{xy}(F) = 0$.

Chapter 4: The Skolem Problem

4.1 Introduction

4.1.1 The Skolem Problem and the Positivity Problem

If $r = (r_n)_{n\geq 0} = (r_0, r_1, r_2, \ldots)$ is a sequence in a ring R and there are constant coefficients $q_1, \ldots, q_k \in R$ with $q_k \neq 0$ such that $r_n = q_1 r_{n-1} + q_2 r_{n-2} + \cdots + q_k r_{n-k}$ for all $n \geq k$, then r is said to be a *linear recurrence of order k in R* with coefficients $q_1, \ldots, q_k$. The equation that r_n satisfies is called a *recurrence relation*. Note that fixing the values of the *initial terms* $r_0, \ldots, r_{k-1}$ of r determines every term r_n inductively, as for all $n \geq k$, the previously computed values of $r_{n-1}, \ldots, r_{n-k}$ can be substituted into the recurrence relation to obtain r_n.

In this chapter, we study two problems on linear recurrences, the *Skolem Problem* and the *Positivity Problem*.

The *Skolem Problem* is the following question: "Given the coefficients and initial terms of a linear recurrence r_n, does there exist a term r_n such that $r_n = 0$?"

The *Positivity Problem* is the question: "Given the coefficients and initial terms of a linear recurrence r_n, do we have $r_n \geq 0$ for all $n \geq 0$?"

The *integer cases* of these problems are to restrict the recurrences to integer coefficients and integer initial terms.

A *positive resolution* to these problems would be to prove that they are decidable, i.e. to prove the existence of an algorithmic decision procedure that would take the coefficients and initial terms as input and give the correct answer to the question as output. A *negative resolution* would be to prove that they are undecidable, that no such decision procedure can exist.

We note that the Positivity problem should perhaps instead be called the *Non-negativity Problem* but its name has been set in the literature [OW12].

4.1.2 History

Perhaps the most well-known linear recurrence is the Fibonacci sequence, the order 2 linear recurrence $(F_n)_{n\geq 0}$ in the integers defined by setting $F_0 = 0$, $F_1 = 1$, and

$F_n = F_{n-1} + F_{n-2}$ for all $n \geq 2$. Leonardo of Pisa, also known as Fibonacci, explicitly introduced this sequence in his *Liber Abaci* of 1202. See [Pis02] for a translation of this work into modern English. This sequence also appears, albeit more cryptically, in earlier works. The earliest known is the *Pingala* in Sanskrit, dating to 450-200 BC [Sin85].

The theory of linear recurrences permeates many fields of mathematics and computer science. The book [EvdPSW03] provides a comprehensive introduction to the vast literature on linear recurrences, summarizing the work contained in over 1300 papers.

The Skolem Problem is viewed as originating in Skolem's paper [Sko35] from 1935. He did not state the problem there in so many words, but rather proved the theorem that the set of indices of the zero terms in a linear recurrence in the rational numbers is the union of a finite set and a finite number of arithmetic progressions. At that time, questions on algorithms had not yet assumed the importance to the research community that they have today, but it is now customary to view the Skolem Problem as a question of decidability [OW12]. Viewed in this light, a positive resolution to the Skolem Problem would be to show that it is a decidable question. This would entail finding or proving the existence of a decision procedure for the question, a well-defined algorithm that takes $q_1, \ldots, q_k$ and $r_0, \ldots, r_{k-1}$ as inputs and, in all cases, takes finite time to correctly decide whether or not there exists an integer n such that $r_n = 0$. A negative resolution would be to find a proof that this is an undecidable question, i.e. to find a proof that no such decision procedure can exist.

As mentioned, the first result on the Skolem Problem is Skolem's 1935 proof of the following theorem for the field of rational numbers.

Theorem 4.1 (Skolem-Mahler-Lech Theorem)**.** *Let k be a field of characteristic 0 and let $r = (r_n)_{n \geq 0}$ be a linear recurrence in k. Then the zero set of r, i.e. $\{n \geq 0 : r_n = 0\}$ is the union of a finite set and a finite number of arithmetic progressions.*

Mahler quickly extended Skolem's result on the rationals to the field of algebraic numbers in [Mah35]. Lech proved the general characteristic 0 case in 1953 in [Lec53]. These results are all ineffective in the sense that they do not give an algorithmic procedure to determine the zero sets. However, in a breakthrough 1976 result, Berstel and Mignotte gave an algorithmic approach to determine all the arithmetic progressions that make up the zero set [BM76]. See [Han86] for an elementary demonstration of this algorithm.

In 2002, Blondel and Portier proved the decidability of the rational case is NP-hard [BP02], which can be viewed as one measure of how hard it will be to solve the Skolem Problem. Their result was for integer linear recurrences, but it is a folklore result that the rational case reduces to the integer case. See Section 4.3.

Partial progress has been achieved by restricting the question to linear recurrences of a fixed order. The decidability of the order 1 case is trivial. The decidability of

the order 2 case is relatively straightforward and is considered folklore. The first real progress on low orders was achieved in the 1980s by Mignotte, Shorey, and Tjideman [MST84] and, independently, Vereshchagin [Ver85], who proved the decidability of the problem for order 3 and order 4 recurrences. The proofs in these papers are complex and deep, using p-adic techniques, Galois theory, and versions of Baker's Theorem on linear forms in logarithms (which earned Baker a Fields Medal in 1970).

The fact that the Skolem Problem has remained open for over 80 years has been described by Terence Tao as "faintly outrageous" [Tao07] and by Richard Lipton as a "mathematical embarrassment" [Lip09]. These comments might be viewed as expressing the hope that there is further progress to be made via an approach that cleverly circumvents the deep mathematics that has seemed necessary to make progress so far.

It is a folklore result that a positive resolution to the Positivity Problem would also give a positive resolution to the Skolem Problem. Indeed, as we will demonstrate in Section 4.3, there is an explicit algorithm to transform any order k instance of the Skolem Problem into an equivalent instance of the Positivity Problem on an integer sequence of order $k^2 + 1$ or less. Thus the Positivity Problem is also viewed has having been open as long as the Skolem problem [OW14]. However, the earliest explicit references to the Positivity Problem date back to the 1970s in [BM76], [Sal76], and [Soi76].

As with the Skolem Problem, the rational case of the Positivity Problem reduces to the integer case (see Section 4.3). As noted in [BDJB10], the NP-hardness result for the Skolem Problem on integer linear recurrences given in [BP02] translates to a proof of the co-NP hardness of the Positivity Problem on integer linear recurrences.

Progress on low order cases of the Positivity Problem is more preliminary than that in the Skolem Problem. In 2006, Halava, et. al, showed that the order 2 case is decidable [HHH06]. In 2009, Laohakosol and Tangsupphathawat showed the order 3 case is decidable [LT09]. These proofs are based on elementary estimates on the roots of the degree k polynomial characteristic equation of an order k linear recurrence, using the formulas for such solutions in terms of radicals. As such, these methods cannot be extended past the order 4 linear recurrences as polynomial equations of degree greater than or equal to 5 have no general solution in radicals. However, Ouaknine and Worrell showed all order 5 or less cases are decidable [OW14]. This was a breakthrough result that used many sophisticated number theoretic techniques. They note that resolving an order 6 or higher case would entail major breakthroughs in the field of Diophantine approximation of transcendental numbers [OW14].

4.1.3 Outline of the Chapter

We organize this chapter as follows. We lay out some fundamental definitions and results that we will need in Section 4.2. In Section 4.3 we will reduce all of our

decision problems to the fundamental problem of deciding the non-negativity of the series coefficients of a rational function. In Section 4.4 we will introduce Type F characteristic polynomials and show how it is possible to decide the non-negativity of rational functions with these denominators. We will give a partial decision procedure for the Skolem Problem and Positivity Problem for recurrences with Type F characteristic polynomials in the final section, Section 4.5. There we will also show that unfortunately this is not a complete decision procedure. We will show that it will not terminate if the characteristic polynomial is not Type F and that there are many such polynomials.

4.2 Fundamentals

4.2.1 Sequences, Series, and Polynomials

We mostly follow the notations and conventions for sequences, polynomials, series, and recurrences given in Stanley's book, [Sta12]. However, for the convenience of the reader, we organize and outline them here as well. We do adopt some notation that is distinct from Stanley's. In particular, we use $\star$ for convolution, $\cdot$ for the Hadamard product, and introduce our own notation for the classes of linear recurrences we will consider.

Let $\mathbb{N} = \{0, 1, 2, \ldots\}$ be the natural numbers, including 0. We write $\mathbb{Z}$, $\mathbb{Q}$, $\mathbb{R}$, and $\mathbb{C}$ for the ring of integers and fields of rational, real, and complex numbers, respectively.

Let R be a ring. A *sequence* in R is a function $r : \mathbb{N} \to R$. The *nth term* of the sequence r is $r(n)$ or, as we sometimes denote it, r_n. We will often write the sequence r as $(r(n))_{n\geq 0}$ or $(r_n)_{n\geq 0}$.

Let R^∞ be the set of sequences in R. We say two sequences p and q in R^∞ are equal if and only if $p(n) = q(n)$ for all $n \in \mathbb{N}$. We let R act on R^∞ via the action $r \cdot (r_n)_{n\geq 0} = (r \cdot r_n)_{n\geq 0}$. We define addition on R^∞ by

$$(p_n)_{n\geq 0} + (q_n)_{n\geq 0} = (p_n + q_n)_{n\geq 0}.$$

R^∞ is an R module with respect to these two operations. The 0-*element* of R^∞ is the 0-*sequence* $0 = (0)_{n\geq 0}$. If $r_1, \ldots, r_k \in R$ and $p_1 = (p_{1n})_{n\geq 0}, \ldots, p_k = (p_{kn})_{n\geq 0}$ are sequences in R^∞, then the *R-linear combination of* $p_1, \ldots, p_k$ *with weights* $r_1, \ldots, r_k$ is the sequence

$$r = r_1 p_1 + \cdots + r_k p_k$$

defined by

$$r(n) = \sum_{j=1}^{k} r_j p_j(n), \text{ for all } n \geq 0.$$

A *formal power series in the indeterminate x with coefficients in R* is the formal symbol

$$r(x) = \sum_{n\geq 0} r_n x^n$$

where $r = (r_n)_{n\geq 0}$ is a sequence in R^∞. Let $R[[x]]$ denote the set of all such series. The 0-*series* $0 = \sum_{n\geq 0} 0x^n$ is the 0-*element* of $R[[x]]$. If $r \in R$ then the power series $r + \sum_{n\geq 1} 0x^n$ is called a *constant* power series. We often denote this series just by r. Thus 0 is a notation for the 0-element. If $r(x) = \sum_{n\geq 0} r_n x^n$, we say that r_0 is the *constant term* of $r(x)$. In general we write $[x^n]r(x)$ for r_n, the coefficient of x^n in $r(x)$. Thus $[x^n] : R[[x]] \to R$ is an R-linear map.

We will often use the maps $G : R^\infty \to R[[x]]$ and $C : R[[x]] \to R^\infty$ defined by

$$G((r_n)_{n\geq 0}) = \sum_{n\geq 0} r_n x^n$$

and

$$C\left(\sum_{n\geq 0} r_n x^n\right) = (r_n)_{n\geq 0}.$$

Clearly these maps are R-linear maps, are bijections, and are inverses of each other. If $r = (r_n)_{n\geq 0}$ is a sequence in R^∞ then

$$g_r(x) = G(r) = \sum_{n\geq 0} r_n x^n$$

is the *generating function of r in $R[[x]]$*. As already noted, the map $g_{(\cdot)}(x)$ is R-linear, i.e.

$$g_{ar+bs}(x) = ag_r(x) + bg_s(x)$$

for all $a, b \in R$ and all $r, s \in R^\infty$. If $r(x) = \sum_{n\geq 0} r_n x^n$ is a power series in $R[[x]]$ then

$$C(r(x)) = (r_n)_{n\geq 0}$$

is the *sequence of coefficients of $r(x)$*, a sequence in R^∞.

We define the action of R on $R[[x]]$ via $r \cdot r(x) = G(r \cdot C(r(x)))$ and the operation of addition in $R[[x]]$ via $r(x) + s(x) = G(C(r(x)) + C(s(x)))$. It is straightforward to show that $R[[x]]$ is an R-module with respect to these operations. The notation we will usually use for these operations is standard,

$$r \cdot r(x) = r \cdot \sum_{n\geq 0} r_n x^n = \sum_{n\geq 0} r \cdot r_n x^n$$

and

$$r(x) + s(x) = \sum_{n\geq 0} r_n x^n + \sum_{n\geq 0} s_n x^n = \sum_{n\geq 0} (r_n + s_n) x^n.$$

Thus $\sum_{n\geq 0} r_n x^n = \sum_{n\geq 0} s_n x^n$ if and only if $r_n = s_n$ for all $n \in \mathbb{N}$.

If $r_1, \ldots, r_k \in R$ and $p_1(x) = \sum_{n\geq 0} p_{1n}x^n, \ldots, p_k(x) = \sum_{n\geq 0} p_{kn}x^n \in R[[x]]$, then the *$R$-linear combination of $p_1(x), \ldots, p_k(x) \in R[[x]]$ with weights $r_1, \ldots, r_k$* is

$$r_1 p_1(x) + \cdots + r_k p_k(x) = \sum_{j=1}^{n} r_k p_k(x) = \sum_{n\geq 0} (r_1 p_{1n} + \cdots r_k p_{kn}) x^n.$$

A sequence r in R^∞ is *finitely supported* or *eventually* 0 if and only if there exists $N \in \mathbb{N}$ such that $r_n = 0$ for all $n > N$. Equivalently, this means $r_n \neq 0$ only finitely often. Let R_0^∞ be the set of finitely supported sequences in R^∞. Clearly when R_0^∞ is viewed as a subset of R^∞ it is closed under the action by R and the operation of addition. Thus R_0^∞ is a submodule of R^∞.

A power series $g_r(x) = \sum_{n\geq 0} r_n x^n \in R[[x]]$ with coefficient sequence $r = (r_n)_{n\geq 0}$ is said to be a *polynomial* if and only if r is a finitely supported sequence, say $r_n = 0$ for $n > N$. We then say that $g_r(x) = \sum_{n=0}^{N} r_n x^n$ is a *polynomial in the indeterminate x with coefficients in R* or, more simply a *polynomial* if x and/or R are understood. We let $R[x]$ be the *ring of polynomials in indeterminate x with coefficients in R*. $R[x]$ is a submodule of $R[[x]]$.

Given a polynomial $r(x) = \sum_{n\geq 0} r_n x^n \in R[x]$ we define the *degree of $r(x)$ in x* to be $\deg(r(x)) = -\infty$ if $(r_n)_{n\geq 0}$ is the 0-sequence and $\deg(r(x)) = \max\{n \in \mathbb{N} : r_n \neq 0\}$ otherwise. As is well-known,

$$\deg(r(x) + s(x)) \leq \max(\deg(r(x)), \deg(s(x))).$$

If R has no zero-divisors, $\deg(ar(x)) = \deg(r(x))$ for all $a \neq 0$ and, more generally,

$$\deg(r(x)s(x)) = \deg(r(x)) + \deg(s(x)).$$

If we adopt the convention that $n + -\infty = -\infty$ for all $n \in \mathbb{N}$, then these formulas remain true even if $r(x)$ or $s(x)$ are the zero polynomial. For convenience in writing proofs with polynomials $r(x)$, we identify the formal expressions $r(x) = \sum_{n\geq 0} r_n x^n = \sum_{n=0}^{N} r_n x^n$ for all $N \geq \deg(r(x))$.

Given a series $r(x) = \sum_{n\geq 0} r_n x^n \neq 0$ we define the *min-degree* $\min\deg(r(x))$ to be the minimum value of k such that $r_k \neq 0$. Note that $\min\deg(r(x)s(x)) = \min\deg(r(x)) + \min\deg(s(x))$.

We adopt all the customary conventions and notations for elements and operations in $R[x]$. The *0-polynomial* is the polynomial $0 = \sum_{n=0} 0x^n$. Two polynomials $\sum_{n=0}^{N} r_n x^n$ and $\sum_{n\geq 0} s_n x^n$ are equal if and only if $r_n = s_n$ for all n with $0 \leq n \leq N$. If $r \in R$ then

$$r \cdot \sum_{n=0}^{N} r_n x^n = \sum_{n=0}^{N} r \cdot r_n x^n.$$

We set

$$\sum_{n=0}^{N} r_n x^n + \sum_{n=0}^{N} s_n x^n = \sum_{n=0}^{N} (r_n + s_n) x^n.$$

If $r_1, \ldots, r_k \in R$ and $p_1(x) = \sum_{n=0}^{N} p_{1n} x^n, \ldots, p_k(x) = \sum_{n=0}^{N} p_{kn} x^n$ then the *R-linear combination of* $p_1(x), \ldots, p_k(x)$ *with weights* $r_1, \ldots r_k$ is the polynomial

$$r_1 p_1(x) + \cdots r_k p_k(x) = \sum_{j=1}^{k} r_j p_j(x) = \sum_{n=0}^{N} (\sum_{j=1}^{k} r_j p_{jn}) x^n = \sum_{n=0}^{N} (r_1 p_{1n} + \cdots + r_k p_{kn}) x^n.$$

If $r = (r_n)_{n \geq 0}$ and $s = (s_n)_{n \geq 0}$ are two sequences in R^∞, we define the *convolution* of r and s to be the sequence $t = r \star s = (t_n)_{n \geq 0}$ in R^∞ given by

$$t_n = \sum_{k=0}^{n} r_k s_{n-k}$$

for all $n \geq 0$. Clearly the convolution of two finitely supported sequences is another finite supported sequence so R_0^∞ is closed under convolution. If R is commutative, we have that $\star$ is a bilinear operation: if $a_1, \ldots, a_k, b_1, \ldots, b_\ell \in R$, $r_1, \ldots, r_k, s_1, \ldots, s_\ell \in R^\infty$, then $(\sum_{i=1}^{k} a_i r_i) \star (\sum_{j=1}^{\ell} b_j s_j) = \sum_{i=1}^{k} \sum_{j=1}^{\ell} a_i b_j (r_i \star s_j)$.

If $r(x) = \sum_{n \geq 0} r_n x^n$ and $s(x) = \sum_{n \geq 0} s_n x^n$ are two power series in $R[[x]]$, we define their *product* to be the power series $t(x) = r(x)s(x) = \sum_{n \geq 0} t_n x^n$ where the sequence $(t_n)_{n \geq 0}$ is the convolution of the sequences $(r_n)_{n \geq 0}$ and $(s_n)_{n \geq 0}$. Thus we write

$$r(x)s(x) = \sum_{n \geq 0} (\sum_{k=0}^{n} r_k s_{n-k}) x^n.$$

Note that for generating functions we have $g_{r \star s}(x) = g_r(x) g_s(x)$.

Note that the result of the action of r on the power series $r(x) = \sum_{n \geq 0} r_n x^n$ defined as $r \cdot r(x) = \sum_{n \geq 0} r \cdot r_n x^n$ is also the product of the constant series r and $r(x)$. Since R_0^∞ is closed under convolution, $R[x]$ is closed under taking products.

From now on we assume that R is a commutative ring with identity. Most often, we will consider the rings $\mathbb{Z}$, $\mathbb{Q}$, $\mathbb{R}$, $\mathbb{C}$. It is a standard theorem that if R is a commutative ring, then $R[[x]]$ and $R[x]$ are commutative rings with respect to addition (defined before) and this product, see [Sta12].

It is a standard fact in algebra that if R is commutative and $x_0 \in R$, then the *substitution map* $\phi_{x_0} : R[x] \to R$ given by $\phi_{x_0}(r(x)) = r(x_0) = \sum_{n=0}^{N} r_n x_0^n$ (where $r(x) = \sum_{n=0}^{N} r_n x^n$) is a ring homomorphism.

Recall that the members of $R[[x]]$ are defined to be formal objects. Even if R is a subring of $\mathbb{C}$, we are generally not concerned with any issues of the convergence of a specific power series in $R[[x]]$ at a specific value of $x \in R$. However, if $r(x) = \sum_{n \geq 0} r_n x^n$

we will find it convenient to write $r(0) = r_0$. Note that, with this convention, if $t(x) = r(x)s(x)$ then $t_0 = t(0) = r(0)s(0) = r_0 s_0$ and so the substitution map $\phi_0(r(x)) = r(0) = r_0$ is a ring homomorphism from $R[[x]]$ to R.

We will sometimes write infinite sums of elements in $R[[x]]$. We will only do this if the sum is *well-defined*, i.e. the process for determining the nth coefficient of the sum is finite.

For example, if $r(x) = \sum_{n\geq 0} r_n x^n$ and $p(x) = \sum_{n\geq 0} p_n x^n$ with $p_0 = p(0) = 0$ we will write $s(x) = r(p(x)) = \sum_{n\geq 0} r_n p^n(x)$. We have that $s(x)$ is well-defined. Since $p(0) = 0$, we have $\min\deg p(x) \geq 1$ and so $\min\deg p^k(x) \geq k(\min\deg p(x)) \geq k$. Thus we have $[x^n]p^j(x) = 0$ for all $j > n$ and so $[x^n]\sum_{j>n} r_j p^j(x) = 0$. Thus

$$[x^n]s(x) = [x^n]\sum_{j=0}^{n} r_j p^j(x)$$

is determined by a finite number of additions and multiplications in R involving the coefficients of $r(x)$ and $p(x)$.

If $p(x) \in R[[x]]$ and $p(0) = 0$ we define the *substitution map* $\phi_{p(x)} : R[[x]] \to R[[x]]$ by $\phi_{p(x)}(r(x)) = r(p(x))$ where $r(p(x))$ is as defined in the previous paragraph.

Theorem 4.2. *Let R be a commutative ring and let $p(x) \in R[[x]]$ with $p(0) = 0$. Then the substitution map $\phi_{p(x)} : R[[x]] \to R[[x]]$ is a ring homomorphism.*

Proof. It has already been proven that $\phi_{p(x)}$ is a well-defined map. Let $r(x) = \sum_{n\geq 0} r_n x^n$ and $s(x) = \sum_{n\geq 0} s_n x^n$ be two series in $R[[x]]$. Then $\phi_{p(x)}(r(x) + s(x)) = \phi_{p(x)}(\sum_{n\geq 0}(r_n + s_n)x^n) = \sum_{n\geq 0}(r_n + s_n)p^n(x) = \sum_{n\geq 0} r_n p^n(x) + \sum_{n\geq 0} s_n p^n(x) = \phi_{p(x)}(r(x)) + \phi_{p(x)}(s(x))$. Also we have

$$\begin{aligned}
\phi_{p(x)}(r(x)s(x)) &= \phi_{p(x)}\left(\sum_{n\geq 0}\left(\sum_{j=0}^{n} r_j s_{n-j}\right)x^n\right) \\
&= \sum_{n\geq 0}\left(\sum_{j=0}^{n} r_j s_{n-j}\right)p^n(x) \\
&= \sum_{n\geq 0}\sum_{j=0}^{n}(r_j p^j(x))(s_{n-j}p^{n-j}(x)) \\
&= \left(\sum_{n\geq 0} r_n p^n(x)\right)\left(\sum_{n\geq 0} s_n p^n(x)\right) \\
&= \phi_{p(x)}(r(x))\phi_{p(x)}(s(x)).
\end{aligned}$$

□

4.2.2 Reciprocals and The Geometric Series

A *reciprocal* or *multiplicative inverse* of $p(x) \in R[[x]]$ is a power series $q(x)$ such that $q(x)p(x) = p(x)q(x) = 1$. If $q(x)$ is the reciprocal of $p(x)$, then we write $q(x) = 1/p(x)$ or $q(x) = (p(x))^{-1}$.

Theorem 4.3. *Let R be a commutative ring. An element $p(x) \in R[[x]]$ has a reciprocal if and only if $p(0)$ is a unit in R. In this case the reciprocal $q(x) = \sum_{k\geq 0} q_k x^k$ is unique and q_k can be recursively defined by $q_0 = 1/p_0$ and*

$$q_k = -(1/p_0)\sum_{j=1}^{k} p_j q_{k-j}$$

for all $k > 0$.

Proof. Suppose $p(0)$ is not a unit. Then there is no series $q(x)$ such $u(x) = p(x)q(x) = 1$. If there were, then we would have $1 = u(0) = p(0)q(0)$ which would contradict the assumption that $p(0)$ is not a unit.

Suppose now that $p_0 = p(0)$ is a unit. We now show that there is a unique series $q(x)$ such that $u(x) = p(x)q(x) = 1$. If $q(x)$ satisfies that equation, we prove, by induction on k, that $q_k = [x^k]q(x)$ is uniquely defined by the recursive process given in the statement of the theorem.

We must have $1 = u(0) = p(0)q(0) = p_0q_0$. Thus $q_0 = 1/p_0$ is the unique inverse of p_0 in R and q_k is uniquely defined for $k = 0$. Suppose now that for some $k \geq 1$, q_j is uniquely defined for $0 \leq j < k$. We will show that q_k is also uniquely defined by the given formula, completing the proof by induction.

For $k \geq 1$ we have

$$0 = [x^k]u(x) = \sum_{j=0}^{k} p_j q_{k-j} = p_0 q_k + \sum_{j=1}^{k} p_j q_{k-j}.$$

We solve this equation for $q_k = -(1/p_0)\sum_{j=1}^{k} p_j q_{k-j}$. Now for all j with $1 \leq j \leq k$ we have $0 \leq k - j < k$ and so q_{k-j} is uniquely defined. This means that q_k is uniquely defined as well. □

If R is a commutative ring with identity, we adopt the convention that $q^0(x) = 1$ for all $q(x) \in R[[x]]$.

Theorem 4.4 (The geometric series)**.** *If R is a commutative ring with identity, then the reciprocal of $p(x) = 1 - x$ is the series $q(x) = 1/(1-x) = \sum_{n\geq 0} x^n$.*

Proof. By Theorem 4.3 we know that $p(x) = 1 - x$ has a unique reciprocal $q(x)$ since $p(0) = 1$ is a unit. We use the formulas given in that theorem to determine the unique sequence of coefficients of $q(x) = \sum_{n\geq 0} q_n x^n$. We show, by induction on k, that $q_k = 1$ for all k. We have $q_0 = 1/p_0 = 1$. For $k \geq 1$ we have

$$q_k = -(1/p_0)\sum_{j=1}^{k} p_j q_{k-j} = -(1/p_0)p_1 q_{k-1} = q_{k-1} = 1.$$

This follows by the facts that $p_0 = 1$ and $p_1 = -1$ and $p_j = 0$ for $j > 1$.

We can also prove this statement by a "guess and verify" approach. As before, we know from Theorem 4.3 that $p(x)$ has a unique reciprocal. We now set $q(x) = \sum_{n\geq 0} x^n$ and verify that $p(x)q(x) = 1$, thereby proving that $q(x)$ is the reciprocal of $p(x)$. We have

$$u(x) = p(x)q(x) = (1-x)\sum_{n\geq 0} x^n = 1\sum_{n\geq 0} x^n - x\sum_{n\geq 0} x^n$$

since $R[[x]]$ is a ring and hence has the distributive property. Thus

$$u(x) = \sum_{n\geq 0} x^n - \sum_{n\geq 0} x^{n+1} = \sum_{n\geq 0} x^n - \sum_{n\geq 1} x^n = 1.$$

□

Theorem 4.5. *Let R be a commutative ring with identity. If $p(x) \in R[[x]]$ and $p_0 = p(0)$ is a unit then the reciprocal of $p(x)$ is given by the formula*

$$\frac{1}{p(x)} = \frac{1}{p_0}\sum_{n\geq 0}\left(\frac{1}{p_0}\right)^n q^n(x)$$

where $q(x) = -(p(x) - p_0)$.

Proof. We have $p(x) = p_0 - q(x)$ so

$$\frac{1}{p(x)} = \frac{1}{p_0 - q(x)} = \frac{1}{p_0(1 - \frac{1}{p_0}q(x))} = \frac{1}{p_0}\frac{1}{1 - \frac{1}{p_0}q(x)}$$

$$= \frac{1}{p_0}\phi_{\frac{1}{p_0}q(x)}\left(\frac{1}{1-x}\right)$$

$$= \frac{1}{p_0}\phi_{\frac{1}{p_0}q(x)}\left(\sum_{n\geq 0} x^n\right).$$

For the last equality, we use the geometric series expansion for $1/(1-x)$ given in Theorem 4.4. Note that the substitution map $\phi_{Q(x)}$ we used, with $Q(x) = \frac{1}{p_0}q(x)$, is well-defined as $Q(0) = (1/p_0)q(0) = 0$, since $q(0) = -(p(0) - p_0) = 0$.

Applying the substitution map in the last equation, we get

$$\frac{1}{p(x)} = \frac{1}{p_0} \sum_{n \geq 0} \left(\frac{1}{p_0} \right)^n q^n(x)$$

as claimed.

□

Corollary 4.6. *Let R be a commutative ring with identity. Let $p(x) \in R[[x]]$ with $p(0) = 0$. Then we have the following formulas:*

$$\frac{1}{1 - p(x)} = \sum_{n \geq 0} p^n(x),$$

$$\frac{1}{1 + p(x)} = \sum_{n \geq 0} (-1)^n p^n(x).$$

Proof. If one applies Theorem 4.5 to the right-hand sides of the equations above, one gets $q(x) = p(x)$ for the first equation and $q(x) = -p(x)$ for the second. □

4.2.3 Recurrences and Rational Functions

Let R be a commutative ring with identity and let k be a positive integer. A sequence $r = (r_n)_{n \geq 0} \in R^\infty$ is a *linear recurrence of order k* if and only if there are ring elements $q_1, \ldots, q_k$ with $q_k \neq 0$ such that $r_n = q_1 r_{n-1} + q_2 r_{n-2} + \cdots + q_k r_{n-k} = \sum_{j=1}^{k} q_j r_{n-j}$ for all $n \geq k$. This formula for r_n with $n \geq k$ is called a *recurrence relation*. The constants, $q_1, \ldots, q_k$ are called the *coefficients of the recurrence relation*. The terms $r_0, \ldots, r_{k-1}$ of r are called the *initial terms* of the sequence. Note that once the order, the coefficients, and the initial terms of a linear recurrence are specified, the rest of the terms, i.e. r_n for $n \geq k$, are uniquely determined, inductively, by substituting previously computed values $r_{n-1}, \ldots, r_{n-k}$ into the recurrence relation to obtain r_n.

If $q_1, \ldots, q_k \in R$ and $q_k \neq 0$, let

$$L_k(R; q_1, \ldots, q_k)$$

be *the set of linear recurrences of order k in R with coefficients $q_1, \ldots, q_k$*. Let

$$L_k(R) = \bigcup \{ L_k(R; q_1, \ldots, q_k) : q_1, \ldots, q_k \in R, q_k \neq 0 \}$$

be *the set of linear recurrences of order k*. Let

$$L(R) = \bigcup_{k \geq 1} L_k(R)$$

be *the set of linear recurrences*. Let

$$G_k(R, q_1, \ldots, q_k) = \{g_r(x) : r \in L_k(R; q_1, \ldots, q_k)\}$$

be the set of generating functions of linear recurrences in $L_k(R; q_1, \ldots, q_k)$. We define $G_k(R)$ and $G(R)$ analogously.

The *characteristic polynomial* of a linear recurrence r with coefficients $q_1, \ldots, q_k$ with $q_k \neq 0$ is defined to be

$$q_r(x) = 1 - \sum_{j=1}^{k} q_j x^j.$$

Note, that our definition is distinct from the common definition that $p(x) = x^k q(1/x) = x^k - \sum_{j=1}^{k} q_j x^{k-j}$ is the characteristic polynomial. We use our definition as it leads to nicer phrasings of our theorems. We will also write

$$L(R, q(x)) = L_k(R; q_1, \ldots, q_k)$$

for the *set of linear recurrences with characteristic polynomial* $q(x)$ and

$$G(R, q(x)) = G_k(R; q_1, \ldots, q_k)$$

for the corresponding set of generating functions.

Theorem 4.7. *Fix a degree k characteristic polynomial $q(x) = 1 - \sum_{j=1}^{k} q_j x^j \in R[x]$ with $q_k \neq 0$. Then we have the following statements.*

1. *If $r = (r_n)_{n \geq 0}$ is a linear recurrence with characteristic polynomial $q(x)$ and k-tuple $(r_0, \ldots, r_{k-1}) \in R^k$ of initial terms, then there is a unique k-tuple of coefficients $(p_0, \ldots, p_{k-1}) \in R^k$ such that*

$$g_r(x) = \sum_{n \geq 0} r_n x^n = \frac{\sum_{n=0}^{k-1} p_n x^n}{q(x)}.$$

2. *Conversely, if $(p_0, \ldots, p_{k-1}) \in R^k$ is a k-tuple of coefficients, then there is a unique k-tuple $(r_0, \ldots, r_{k-1}) \in R^k$ such that if $r = (r_n)_{n \geq 0}$ is the linear recurrence with characteristic polynomial $q(x)$ and sequence of initial terms $(r_0, \ldots, r_{k-1})$ then, again,*

$$g_r(x) = \sum_{n \geq 0} r_n x^n = \frac{\sum_{n=0}^{k-1} p_n x^n}{q(x)}.$$

3. *The correspondences between k-tuples of initial terms $(r_0, \ldots, r_{k-1}) \in R^k$ and k-tuples of coefficients $(p_0, \ldots, p_k) \in R^k$ described in statements 1 and 2 are bijective linear maps given by*

$$p_i = r_i - \sum_{j=1}^{i} q_i r_{i-j}, \text{ for all } 0 \leq i < k.$$

Proof. Extend the definition of the sequence q_j to $q_j = 0$ for $j \geq k$ so that $q(x) = 1 - \sum_{j \geq 1} q_j x^j$. We have

$$\frac{\sum_{n=0}^{k-1} p_n x^n}{q(x)} = \sum_{n \geq 0} r_n x^n$$

if and only if

$$\sum_{i=0}^{k-1} p_i x^i = (1 - \sum_{j \geq 1} q_j x^j) \sum_{n \geq 0} r_n x^n = \sum_{i \geq 0} (r_i - \sum_{j=1}^{i} q_i r_{i-j}) x^i.$$

Since series can only be equal if they are equal term by term we see that this last statement is true if and only if $r_i = \sum_{j=1}^{i} q_i r_{i-j}$ for $i \geq k$ (i.e. $(r_n)_{n \geq 0}$ is a recurrence with characteristic polynomial $q(x)$) and $p_i = r_i - \sum_{j=1}^{i} q_i r_{i-j}$ for $0 \leq i < k$ (i.e. the equations in statement 3).

Clearly the equations map $(r_0, \ldots, r_{k-1})$ linearly and uniquely to $(p_0, \ldots, p_{k-1})$. They are also invertible. This is because if $(p_0, \ldots, p_{k-1})$ is fixed, then $(r_0, \ldots, r_{k-1})$ is also uniquely determined by the system as we may use the equations in order of increasing i with $0 \leq i \leq k-1$ to recursively solve for the r_i in terms of the p_i. □

If $q(x) \in R[x]$ has $\deg(q(x)) \geq 1$ and $q(0) = 1$, let

$$\mathcal{R}(R, q(x)) = \{p(x)/q(x) : p(x) \in R[x], \deg(p(x)) < \deg(q(x))\}.$$

Theorem 4.7 has the following corollary.

Corollary 4.8. *If $R = \mathbb{Z}$, $\mathbb{Q}$, or $\mathbb{R}$, we have that the sequences in $L(R, q(x))$ are in one-to-one correspondence with the rational functions in $\mathcal{R}(R, q(x))$. The generating function of each sequence $r \in L(R, q(x))$ is a rational function $p(x)/q(x) \in \mathcal{R}(R, q(x))$ and each rational function $p(x)/q(x) \in \mathcal{R}(R, q(x))$ is the generating function of a sequence $r \in L(R, q(x))$.*

Theorem 4.9. *Given a characteristic polynomial $q(x)$, the set of linear recurrences $L(R, q(x))$ is closed under R-linear combinations. The set $L(R)$ of general linear recurrences is closed under linear combinations and convolutions.*

Proof. By Theorem 4.7, if $r, s \in L(R, q(x))$ then there are polynomials $p_r(x), p_s(x)$ of degree less than $\deg(q(x))$ such that

$$g_r(x) = \frac{p_r(x)}{q(x)}, \quad \text{and } g_s(x) = \frac{p_s(x)}{q(x)}.$$

But then, if $a, b \in R$,

$$g_{ar+bs}(x) = ag_r(x) + bg_s(x) = \frac{ap_r(x) + bp_s(x)}{q(x)}.$$

Since $\deg(ap_r(x) + bp_s(x)) < \deg(q(x))$, $ar + bs \in L(R, q(x))$ as well.

Suppose now that $r, s \in L(R)$. Then $r \in L(R, q_r(x))$ and $s \in L(R, q_s(x))$ where $q_r(x)$ and $q_s(x)$ are characteristic polynomials. In particular, we have $q_r(0) = q_s(0) = 1$. Also there are polynomials $p_r(x)$ and $p_s(x)$ with $\deg(p_r(x)) < \deg(q_r(x))$ and $\deg(p_s(x)) < \deg(q_s(x))$ such that

$$g_r(x) = \frac{p_r(x)}{q_r(x)} \text{ and } g_s(x) = \frac{p_s(x)}{q_s(x)}.$$

Let $a, b \in R$. Then

$$g_{ar+bs} = ag_r(x) + bg_s(x) = \frac{ap_r(x)q_s(x) + bp_s(x)q_r(x)}{q_r(x)q_s(x).}$$

Note that $Q(0) = q_r(0)q_s(0) = 1$ so $Q(x) = q_r(x)q_s(x)$ is a characteristic polynomial.

Let $P(x) = ap_r(x)q_s(x) + bp_s(x)q_r(x)$. We have

$$\deg(P(x)) \leq \max(\deg(p_r(x)q_s(x)), \deg(p_s(x)q_r(x))).$$

Note that $\deg(p_r(x)q_s(x)) = \deg(p_r(x)) + \deg(q_s(x)) < \deg(q_r(x)) + \deg(q_s(x)) = \deg(q_r(x)q_s(x)) = \deg(Q(x))$ and similarly $\deg(p_s(x)q_r(x)) < \deg(Q(x))$ as well. Thus $\deg(P(x)) < \deg(Q(x))$ and $ar + bs \in L(R, Q(x))$ is a linear recurrence with characteristic polynomial $Q(x)$.

Let $t = r \star s$ be the convolution of r and s. Then

$$g_t(x) = g_r(x)g_s(x) = \frac{p_r(x)p_s(x)}{Q(x)}.$$

Since $\deg(p_r(x)p_s(x)) = \deg(p_r(x))+\deg(p_s(x)) < \deg(q_r(x))+\deg(q_s(x)) = \deg(Q(x))$ we have that $t \in L(R, Q(x))$. □

4.2.4 The Rational Non-Negativity Problem

If $p(x) = \sum_{n\geq 0} p_n x^n$ and $q(x) = \sum_{n\geq 0} q_n x^n$ are two series in $\mathbb{R}[[x]]$, we say $p(x)$ is *dominated by* $q(x)$ if and only if $p_n \leq q_n$ for all $n \geq 0$. We denote this by $p(x) \sqsubseteq q(x)$ or $q(x) \sqsupseteq p(x)$. Alternatively, we say $q(x)$ *dominates* $p(x)$.

We say $p(x)$ is *non-negative* if $p_n \geq 0$ for all $n \geq 0$. This is equivalent to $p(x) \sqsupseteq 0 = \sum_{n\geq 0} 0x^n$.

The following lemma describes situations under which non-negativity is closed.

Lemma 4.10. *Let $p(x), q(x) \sqsupseteq 0$. Then we have the following statements.*

1. *For all $a, b \in \mathbb{R}$ with $a, b \geq 0$ we have $ap(x) + bq(x) \sqsupseteq 0$*
2. *We have $p(x)q(x) \sqsupseteq 0$.*
3. *If $q(0) = 0$ we have $p(x)/(1 - q(x)) \sqsupseteq 0$.*

Proof. The first statement is trivial. For the second, note that $p(x)q(x) = \sum_{n\geq 0} t_n x^n$ where $t_n = \sum_{k=0}^{n} p_k q_{n-k} \geq 0$.

For the third statement, note that by Corollary 4.6 we have

$$s(x) = \frac{p(x)}{1 - q(x)} = p(x) \sum_{n\geq 0} q^n(x) = \sum_{n\geq 0} p(x)q^n(x).$$

Since $\min\deg(q(x)) \geq 1$, $\min\deg p(x)q^n(x) = \min\deg(p(x)) + n \cdot \min\deg(q(x)) \geq n$. Thus $[x^k]p(x)q^n(x) = 0$ for $n > k$. Thus we have that the coefficient $[x^k]t(x) = [x^k]\sum_{n=0}^{k} p(x)q^n(x)$. But $p(x)q^k(x)$ is non-negative for all k, so the coefficient in question is non-negative. □

With this in mind we introduce the *Rational Non-Negativity Problem.*

The *Rational Non-Negativity Problem* is to decide, when given $p(x), q(x) \in \mathbb{Q}[x]$ with $q(0) = 1$ whether $p(x)/q(x) \sqsupseteq 0$. The *integer case* of this problem is the restriction of the polynomials to $\mathbb{Z}[x]$.

Theorem 4.11. *The Positivity Problem reduces to the Rational Non-Negativity problem. This is also true for the integer cases of these problems.*

Proof. If there is a decision procedure for the Rational Non-Negativity Problem we can use it to design a decision procedure for the Positivity Problem. Suppose a linear recurrence r in $\mathbb{Q}$ (or $\mathbb{Z}$) is given with characteristic polynomial $q(x) \in \mathbb{Q}[x]$ (or $\mathbb{Z}[x]$). Then by Theorem 4.7 and Corollary 4.8 we can explicitly determine a polynomial $p(x) \in \mathbb{Q}[x]$ (or $\mathbb{Z}[x]$) so that the generating function for r is $p(x)/q(x)$. The non-negativity of r is thus equivalent to the non-negativity of $p(x)/q(x)$. We can then answer this via the hypothetical decision procedure we have for these rational functions. □

4.3 Reductions

4.3.1 Reduction to the Integer Case

The following reduction is often stated in the folklore of the Skolem and Positivity Problems.

Theorem 4.12. *The Skolem and Positivity Problems reduce to their integer cases.*

Proof. Given a linear recurrence $(r_n)_{n\geq 0}$ in $\mathbb{Q}$, there is a positive integer B such that the sequence $R_n = B^{n+1}r_n$ is a recurrence in $\mathbb{Z}$ with coefficients in $\mathbb{Z}$.

Indeed, suppose $q_1 = a_1/b_1, \ldots, q_k = a_k/b_k$ and $r_0 = A_0/B_0, \ldots, r_{k-1} = A_{k-1}/B_{k-1}$ are rational numbers where a_i, b_i, A_i, B_i are integers with $b_i, B_i > 0$ and suppose

$$r_n = \sum_{i=1}^{k} q_i r_{n-i}$$

for $n \geq k$.

Let $B = \prod_{i=1}^{k} b_i \prod_{j=0}^{k-1} B_j$. Then we have

$$R_n = B^{n+1}r_n = \sum_{i=1}^{k} B^i q_i B^{n-i+1} r_n = \sum_{i=1}^{k} B^i q_i R_{n-i}.$$

The coefficients $q_i B^i = a_i B^i/b_i$ of this recurrence are integers as $i \geq 1$ and b_i is a factor of B. The initial terms $R_i = B^{i+1}r_i = B^{i+1}A_i/B_i$ are also integers as $i \geq 0$ and B_i is a factor of B.

If we have a decision procedure for the integer case, we can transform an input rational recurrence r_n to an integer recurrence $R_n = B^{n+1}r_n$ and apply the decision procedure to that. Since $r_n = 0$ if and only if $R_n = 0$ and $r_n \geq 0$ if and only if $R_n \geq 0$ we will have also decided the Skolem Problem or Positivity Problem for r_n.

□

4.3.2 Closure Under The Hadamard Product

Given two linear recurrences $r = (r_n)_{n\geq 0}$ and $s = (s_n)_{n\geq 0}$ in a ring R, the *Hadamard product* of r and s is the sequence $t = (t_n)_{n\geq 0}$ defined by $t_n = r_n s_n$ for $n \geq 0$. See [Sta12]. We write $t = r \cdot s$ for the Hadamard product. We have the following closure result on the Hadamard product.

Theorem 4.13 (Hadamard Products of Complex Recurrences)**.** *If r and s are linear recurrences in $\mathbb{C}$ of orders k and ℓ then the Hadamard product $t = r \cdot s$ is a linear recurrence in $\mathbb{C}$ of order $k\ell$ or less.*

The standard proof of this theorem uses the following result, found in [Sta12].

Theorem 4.14. *Let $q(x) = 1 - \sum_{i=1}^{k} q_i x^i \in \mathbb{C}[x]$ with $q_k \neq 0$ be a characteristic polynomial of degree k. Let $\gamma_1, \ldots, \gamma_\ell$ with $1 \leq \ell \leq k$ be distinct non-zero complex numbers and let $m_1, \ldots, m_\ell$ be positive integers with $\sum_{i=1}^{\ell} m_i = k$ such that*

$$q(x) = \prod_{i=1}^{\ell} (1 - \gamma_i x)^{m_i}.$$

Then r_n is a linear recurrence in $\mathbb{C}$ with characteristic polynomial $q(x)$ if and only if there exist polynomials $p_1(x), \ldots, p_\ell(x) \in \mathbb{C}[x]$ with $\deg(p_i(x)) < m_i$ such that

$$r_n = \sum_{i=1}^{\ell} p_i(n) \gamma_i^n$$

for all $n \geq 0$.

Proof. (of Theorem 4.13) Let $q(x) = \prod_{i=1}^{\ell} (1 - \gamma_i x)^{m_i}$ be the characteristic polynomial for the recurrence r and let $Q(x) = \prod_{j=1}^{L} (1 - \Gamma_j x)^{M_j}$ be the characteristic polynomial for s. Then by Theorem 4.14,

$$r_n = \sum_{i=1}^{\ell} p_i(n) \gamma_i^n$$

and

$$s_n = \sum_{j=1}^{L} P_j(n) \Gamma_j^n$$

for all $n \geq 0$ where we have $\deg(p_i(x)) < m_i$ for $1 \leq i \leq \ell$ and $\deg(P_j(x)) < M_j$ for $1 \leq i \leq L$. Thus

$$t_n = r_n s_n = \sum_{i,j} p_i(n) P_j(n) (\gamma_i \Gamma_j)^n.$$

Let $h(x) = \prod_{i,j} (1 - \gamma_i \Gamma_j x)^{m_i M_j}$. Note that $\deg(p_i(x) P_j(x)) = \deg(p_i(x)) + \deg(P_j(x)) \leq (m_i - 1) + (M_j - 1) \leq m_i M_j - 1$. Note that the last inequality is equivalent to $(m_i - 1)(M_i - 1) \geq 0$ which is true since $m_i, M_j \geq 1$. Thus Theorem 4.14 tells us that t_n is a linear recurrence in $\mathbb{C}$ with characteristic polynomial $h(x)$. Note that $\deg(h(x)) = \sum_{i,j} m_i M_j = \sum_i m_i \sum_j M_j = \deg(q(x)) \deg(Q(x))$.

□

We have the following theorem.

Theorem 4.15 (Hadamard Products of Integer Recurrences)**.** *If r and s are linear recurrences in $\mathbb{Z}$ of orders k and ℓ, then the Hadamard product $t = r \cdot s$ is a linear recurrence in $\mathbb{Z}$ of order $k\ell$ or less.*

This is often stated without proof in the literature and it is surprisingly difficult to find a proof. We failed to find one. There is an outline of a proof in [Yua] but it is missing the complete proof of significant details. We build that argument into a complete proof here.

Proof. Set the same notation as in the proof of Theorem 4.13. Given a characteristic function $q(x) = 1 - \sum_{i=1}^{k} q_i x^i$, we form the $k \times k$ *companion matrix* $M(q)$ defined by

$$M(q) = \begin{bmatrix} 0 & 1 & & & \\ & 0 & \ddots & & \\ & & \ddots & 1 & \\ & & & 0 & 1 \\ q_k & q_{k-1} & \cdots & q_2 & q_1 \end{bmatrix}$$

All missing matrix entries in the above definition are taken to be 0's. It is easy to prove $q(x) = \det(I - M(q)x)$ by induction on k.

Since the roots of $q(x) = \prod_{i=1}^{\ell}(1 - \gamma_i x)^{m_i}$ are $1/\gamma_i$ with multiplicity m_i each γ_i is an eigenvalue of $M(q)$ of multiplicity m_i. Indeed, note that $\det(I - M(q)(1/\gamma_i)) = 0$ implies that $I - M(q)(1/\gamma_i)$ is singular and there is a non-zero vector v such that $(I - M(q)(1/\gamma_i))v = 0$, or $M(q)v = \gamma_i v$.

The matrix $M(q)$ is similar to a matrix $J(q) = A(q)M(q)A^{-1}(q)$ in Jordan canonical form. This matrix $J(q)$ will be upper triangular with m_i copies of γ_i for each $1 \le i \le \ell$ down its diagonal. Similarly $J(Q) = A(Q)M(Q)A^{-1}(Q)$ will have M_j copies of Γ_j for each $1 \le j \le L$ down its diagonal. Consider $h(x) = \det(I - M(q) \otimes M(Q)x)$ where $\otimes$ is the matrix tensor product.

We have

$$\begin{aligned} h(x) &= \det((A(q) \otimes A(Q))(I - M(q) \otimes M(Q)x)(A(q) \otimes A(Q))^{-1}) \\ &= \det(I - (A(q) \otimes A(Q))(M(q) \otimes M(Q))(A^{-1}(q) \otimes A^{-1}(Q)x) \\ &= \det(I - (A(q)M(q)A^{-1}(q) \otimes A(Q)M(Q)A^{-1}(Q)x) \\ &= \det(I - J(q) \otimes J(Q)x) \end{aligned}$$

Since $J(q) \otimes J(Q)$ will be upper triangular and will have $m_i M_j$ copies of $\gamma_i \Gamma_j$ down its diagonal for $1 \le i \le \ell$ and $1 \le j \le L$, $h(x) = \prod_{i,j}(1 - \gamma_i \Gamma_j x)^{m_i M_j}$.

The proof of Theorem 4.13 tell us that $t = r \cdot s$ is a linear recurrence with characteristic polynomial $h(x)$. Since $q(x)$ and $Q(x)$ are integer polynomials, $M(q)$

and $M(Q)$ are integer matrices and $h(x) = \det(I - M(q) \otimes M(Q)x)$ is an integer polynomial. Thus the recurrence relation that t satisfies has integer coefficients, the coefficients of $h(x)$. Since r_n and s_n are integer sequences then so is t_n, and so t_n is an integer linear recurrence. □

4.3.3 Reduction to the Integer Case of the Positivity Problem

The following reduction is also often asserted in the literature, without a full proof.

Theorem 4.16. *The integer case of the Skolem Problem reduces to the integer case of the Positivity Problem.*

Proof. Let $(r_n)_{n\geq 0}$ be an integer linear recurrence of order k. We construct another integer linear recurrence $(R_n)_{n\geq 0}$, given by $R_n = r_n^2 - 1$. Then clearly $R_n = -1 < 0$ if and only if $r_n = 0$. So $(r_n)_{n\geq 0}$ never has a zero if and only if $(R_n)_{n\geq 0}$ is always non-negative.

Since r_n^2 is the Hadamard product of r_n with itself, the proof of Theorem 4.15 constructs $h(x) \in \mathbb{Z}[x]$ of degree k^2 that is the characteristic polynomial for r_n^2. Theorem 4.7 shows us how to find the coefficients of a function $p(x) \in \mathbb{Z}[x]$ such that the generating function for r_n^2 is $p(x)/h(x)$.

Theorem 4.4 tells us that the generating function of the constant sequence 1 is $1/(1-x)$. So the generating function for $R_n = r_n^2 - 1$ is $p(x)/h(x) - 1/(1-x) = P(x)/(h(x)(1-x))$ where $P(x) = p(x)(1-x) - h(x)$.

Since $\deg(P(x)) \leq \deg(h(x)) < \deg(h(x)(1-x))$ we have by Theorem 4.7 that $R_n = r_n^2 - 1$ is an integer linear recurrence of order $\deg(h(x)(1-x)) = k^2 + 1$ or less.

Now the decision procedure for the Positivity Problem on $(R_n)_{n\geq 0}$ can be used to decide the Skolem Problem for $(r_n)_{n\geq 0}$. □

4.3.4 Reduction to the Rational Non-Negativity Problem

We have the following Theorem.

Theorem 4.17. *The Skolem Problem and the Positivity Problem reduce to the Integer Case of the Rational Non-Negativity Problem.*

Proof. We saw that the rational cases of the Skolem Problem and Positivity Problem reduced to the integer cases in Theorem 4.12. We saw that the integer case of the Skolem Problem reduces to the integer case of the Positivity Problem in Theorem 4.16. We saw that the integer case of the Positivity Problem reduces to the integer case of Rational Non-Negativity in Theorem 4.11. □

4.4 Type F Polynomials

Here is a quick proof that $1/q(x) = 1/(1-x+x^3-x^4)$ is non-negative. $(1+x+x^2)q(x) = 1-x^6$ and so

$$1/q(x) = (1+x+x^2)/(1-x^6).$$

By Lemma 4.10, since $P(x) = 1+x+x^2$ and $Q(x) = x^6$ are both non-negative, we have $1/q(x) = P(x)/(1-Q(x))$ is non-negative.

With this as motivation, we define a characteristic polynomial to be *Type F* if and only if there are polynomials $f(x), Q(x) \in \mathbb{Q}[x]$, with $f(0) = 0$ and $Q(0) = 0$, $Q(x)$ not identically zero, and $Q(x)$ non-negative so that $(1+f(x))q(x) = 1-Q(x)$. We say that $f(x), Q(x)$ *witness* that $q(x)$ is Type F. The following theorem will immediately show the importance of $q(x)$ being type F.

Theorem 4.18. *Let $q(x) \in \mathbb{Q}[x]$ with $q(0) = 1$. If $q(x)$ is type F, with witnessing polynomials $f(x), Q(x) \in \mathbb{Q}[x]$, then, given input $p(x) \in \mathbb{Q}[x]$, it is decidable if $p(x)/q(x)$ is non-negative.*

Before proving this, we need a little notation. Given $x \in \mathbb{R}$, let

$$x^+ = \max(x, 0) \text{ and } x^- = \max(-x, 0)$$

so that $x = x^+ - x^-$, $x^+, x^- \geq 0$ and $x^+ \neq 0$ implies $x^- = 0$ (and vice-vera). We call x^+ and x^- the *positive part* and *negative part* of x. If $p(x) \in \mathbb{R}[[x]]$, let the *positive part* of $p(x)$ be

$$p^+(x) = \sum_{n \geq 0} p_n^+ x^n$$

and the *negative part* be

$$p^-(x) = \sum_{n \geq 0} p_n^- x^n.$$

Thus $p(x) = p^+(x) - p^-(x)$ and $p^+(x)$ and $p^-(x)$ are non-negative. Let $supp(p(x)) = \{n : p_n \neq 0\}$ be the *support* of $p(x)$. So $p^+(x)$ and $p^-(x)$ have disjoint support, i.e. $supp(p^+(x)) \cap supp(p^-(x)) = \emptyset$.

Proof. Suppose $q(x)$ is type F so that there exists $f(x), Q(x) \in \mathbb{Q}[x]$ with $f(0) = 0$ and $Q(0) = 0$ and $Q(x)$ not identically zero and $Q(x)$ non-negative so that $(1+f(x))q(x) = 1-Q(x)$. Suppose now that $p(x) \in \mathbb{Z}[x]$ is given. Then

$$\frac{p(x)}{q(x)} = \frac{P(x)}{1-Q(x)}$$

where $P(x) = (1+f(x))p(x) \in \mathbb{Z}[x]$.

Set $P_0(x) = P(x)$ and find $P_1(x)$ so that we have

$$\frac{P(x)}{1-Q(x)} = \frac{P_0(x)}{1-Q(x)} = P_0^+(x) + \frac{P_1(x)}{1-Q(x)}.$$

Clearing denominators and replacing $P_0(x)$ by $P_0(x) = P_0^+(x) - P_0^-(x)$ we get

$$P_0^+(x) - P_0^-(x) = P_0^+(x)(1-Q(x)) + P_1(x)$$

or

$$P_1(x) = P_0^+(x)Q(x) - P_0^-(x).$$

Recursively repeating this process on $P_i(x)/(1-Q(x))$ for $1 \leq i \leq k$, we get

$$\frac{P(x)}{1-Q(x)} = P_0^+(x) + P_1^+(x) + \cdots + P_k^+(x) + \frac{P_{k+1}(x)}{1-Q(x)}$$

where we have $P_i(x) = P_{i-1}^+(x)Q(x) - P_{i-1}^-(x)$ for all i with $1 \leq i \leq k+1$.

Suppose we have $P_{k+1}^-(x) = 0$ that for some k. Then we have discovered that

$$\frac{P(x)}{1-Q(x)} = P_0^+(x) + P_1^+(x) + \cdots + P_k^+(x) + \frac{P_{k+1}(x)}{1-Q(x)}$$

$$= P_0^+(x) + P_1^+(x) + \cdots + P_k^+(x) + \frac{P_{k+1}^+(x)}{1-Q(x)}$$

Since the $P_i^+(x)$ and $Q(x)$ are all non-negative, this proves that $P(x)/(1-Q(x))$ is non-negative. From now on we will assume that $P_i^-(x) \neq 0$ for all $i \geq 0$.

Suppose now we have $P_{k+1}^+(x) = 0$ for some k. Then we will have discovered that

$$\frac{P(x)}{1-Q(x)} = P_0^+(x) + P_1^+(x) + \cdots + P_k^+(x) + \frac{P_{k+1}(x)}{1-Q(x)}$$

$$= P_0^+(x) + P_1^+(x) + \cdots + P_k^+(x) - \frac{P_{k+1}^-(x)}{1-Q(x)}$$

Since $P_{k+1}^-(x) \neq 0$, $P_{k+1}^-(x)/(1-Q(x))$ has infinitely many positive terms. This shows that $P(x)/(1-Q(x))$ has infinitely many negative terms. Also, since $P_0^+(x) + P_1^+(x) + \cdots + P_k^+(x)$ is a polynomial, $P(x)/(1-Q(x))$ eventually has only negative and zero terms.

From now on we will also assume that $P_i^-(x) \neq 0$ for all $i \geq 0$.

The polynomials $P_i^+(x)$ and $P_i^-(x)$ are non-negative and have disjoint support. Further more $P_i^+(x)Q(x)$ is non-negative as well since $Q(x)$ is non-negative. Thus we have

$$P_i^+(x) \sqsubseteq P_{i-1}^+(x)Q(x) \sqsubseteq P_0^+(x)Q^i(x)$$

and

$$P_i^-(x) \sqsubseteq P_{i-1}^-(x) \sqsubseteq P_0^-(x)$$

for all $i \geq 1$.

Thus

$$\min\deg P_i^+(x) \geq \min\deg P_0^+(x) + i \min\deg Q(x)$$

and

$$\deg(P_i^-(x)) \leq \deg(P_0^-(x))$$

for all $i \geq 1$.

This also gives

$$P_i^-(x) \sqsubseteq P_{i-1}^-(x) \sqsubseteq \cdots \sqsubseteq P_j^-(x)$$

for all i, j with $0 \leq j < i \leq k+1$ and so $P_k^-(x)$ and $P_j^+(x)$ have disjoint support for all $j \leq k$. This means $P_k^-(x)$ and $f_k(x) = P_0^+(x) + P_1^+(x) + \cdots + P_k^+(x)$ have disjoint support.

Take $k \geq 1$ big enough so that $\min\deg P_k(x) \geq \min\deg P_0^+(x) + k \min\deg(Q(x)) > \deg P_0^-(x) \geq \deg P_k^-(x)$. Then we have

$$\begin{aligned}
\frac{P(x)}{1-Q(x)} &= P_0^+(x) + P_1^+(x) + \cdots + P_k^+(x) + \frac{P_{k+1}(x)}{1-Q(x)} \\
&= f_k(x) + \frac{P_k^+(x)}{1-Q(x)} - \frac{P_k^-(x)}{1-Q(x)} \\
&= f_k(x) + \frac{P_k^+(x)}{1-Q(x)} - \frac{P_k^-(x)Q(x)}{1-Q(x)} - P_k^-(x)
\end{aligned}$$

Let

$$\begin{aligned}
g_k(x) &= f_k(x) + \frac{P_k^+(x)}{1-Q(x)} \\
h_k(x) &= \frac{P_k^-(x)}{1-Q(x)} = \frac{P_k^-(x)Q(x)}{1-Q(x)} + P_k^-(x)
\end{aligned}$$

so that

$$\frac{P(x)}{1-Q(x)} = g_k(x) - h_k(x)$$

Since $P_k^-(x) \neq 0$, $[x^d]P_k^-(x) > 0$ where $d = \deg(P_k^-(x))$. Since

$$h_k(x) = \frac{P_k^-(x)Q(x)}{1-Q(x)} + P_k^-(x) \sqsupseteq P_k^-(x)$$

and $P_k^-(x)/(1-Q(x))$ is non-negative $[x^d]h_k(x) > 0$ as well. Since

$$\min\deg P_k^+(x)/(1-Q(x)) \geq \min\deg P_k^+(x) > \deg(P_k^-(x)) = d$$

by choice of k and since $f_k(x)$ and $h_k(x)$ have disjoint support we have $[x^d]g_k(x) = 0$. Thus

$$[x^d]P(x)/(1-Q(x)) = [x^d]g_k(x) - [x^d]h_k(x) < 0$$

and $P(x)/(1-Q(x))$ has been proven to fail to be non-negative. □

We see that Theorem 4.18 implicitly gives the following decision procedure.

Algorithm 4.19. *Rational Non-Negativity Decision for Type F Denominators*

Input: $p(x), q(x) \in \mathbb{Q}[x]$ and $f(x), Q(x) \in \mathbb{Q}[x]$ witnessing $q(x)$ is Type F.

Output: Decision on whether $p(x)/q(x) \sqsupseteq 0$ or not.

1. Compute $P_0(x) = P(x) = (1+f(x))p(x)$, $P_0^+(x)$, and $P_0^-(x)$.
2. For $k = 0, 1, 2 \ldots$
   (a) Compute $P_{k+1}(x) = P_k^+(x)Q(x) - P_k^-(x)$, $P_{k+1}^+(x)$, and $P_{k+1}^-(x)$.
   (b) If $P_{k+1}^-(x) = 0$, then end and return: "Yes. $p(x)/q(x) \sqsupseteq 0$."
   (c) If $P_{k+1}^-(x) \neq 0$, and $P_{k+1}^+(x) = 0$, then end and return: "No. $p(x)/q(x) \not\sqsupseteq 0$, $[x^n]p(x)/q(x) \leq 0$ for $n > d$, and $[x^n]p(x)/q(x) < 0$ infinitely often for $n > d$ where $d = \deg(P_0^+(x) + \cdots + P_k^+(x))$."
   (d) If $P_{k+1}^-(x) \neq 0$, and $P_{k+1}^+(x) \neq 0$ and $\min\deg(P_{k+1}^+(x)) > \deg(P_{k+1}^-(x))$, then end and return "No. $p(x) \geq q(x) \not\sqsupseteq 0$ and $[x^n]p(x)/q(x) < 0$ for $n = \deg(P_k^-(x))$."

We have the following partial decision procedure that, when given a type F polynomial $q(x)$, will return $f(x)$ and $Q(x)$ witnessing this fact, but will not terminate if $q(x)$ is not Type F.

Algorithm 4.20. *Witnesses for Type F Polynomials*

Input: $q(x) = 1 + \sum_{n=1}^{k} q_n x^n \in \mathbb{Q}[x]$ with $q(0) = 1$, $\deg(q(x)) \geq 1$

Output: If $q(x)$ is Type F, $f(x), Q(x) \in \mathbb{Q}[x]$ will be returned witnessing this fact. Otherwise, the algorithm does not terminate.

1. Set $\ell = 1$.
2. Use linear programming to decide if the system of linear inequalities

$$-Q_n = q_n + \sum_{j=1}^{\ell} q_{n-j} f_j \leq 0, \text{ for } 1 \leq n \leq k+\ell$$

is feasible for $f_1, \ldots, f_\ell \in \mathbb{R}$ or not by either producing a rational feasible solution or proving no feasible solution exists. (We take $q_n = 0$ for $n > k$ or $n < 0$ and $q_0 = 1$.)

3. If the system is feasible, end and return "$q(x)$ is Type F with $f(x) = \sum_{n=1}^{\ell} f_n x^n$ and $Q(x) = \sum_{n=1}^{k+\ell} Q_n x^n$ as witnesses."

4. If the system is not feasible, set $\ell \leftarrow \ell + 1$ and go to line 2.

Theorem 4.21. *If $q(x)$ is Type F, Algorithm 4.20 terminates and produces $f(x), Q(x) \in \mathbb{Q}[x]$ with $f(0) = 0$ and $Q(x) = 0$, $Q(x) \sqsupseteq 0$ and $Q(x)$ not identically 0 such that $(1 + f(x)) = 1 - Q(x)$. Otherwise Algorithm 4.20 does not terminate.*

Proof. Let $\ell \geq 1$ and let $f(x) = \sum_{n=1}^{\ell} f_n x^n$. Let $Q(x) = \sum_{n=1}^{k+\ell} Q_n x^n$ with $Q_n \geq 0$ and with some $Q_n \neq 0$. The equation $(1 + f(x))q(x) = 1 - Q(x)$ is equivalent to the condition that $[x^n](1 + f(x))q(x) = [x^n](1 - Q(x))$ for all $1 \leq n \leq k + \ell$ which is the system of linear inequalities in the description of the algorithm. Thus $q(x)$ is of Type F with $f(x)$ of degree ℓ or less if and only if this linear system is feasible. Note that since $\deg(q) \geq 1$ we cannot have $Q(x) = 0$.

If $q(x)$ is Type F, there will be some ℓ for which this linear system is feasible. The algorithm will find a feasible solution and hence produce $f(x)$ and $Q(x)$. Note that the linear inequalities have integer coefficients and constants so a linear programming algorithm like the simplex algorithm with a pivot rule guaranteed to terminate will find a rational feasible point and hence produce $f(x), Q(x) \in \mathbb{Q}[x]$.

If the Algorithm terminates, it will have proved $q(x)$ is Type F. Therefore it cannot terminate if $q(x)$ is not Type F. □

Not all characteristic polynomials are type F as the following theorem shows. Thus Algorithm 4.20 is only a partial decision procedure. It will not terminate for recurrences with those characteristic polynomials.

Theorem 4.22. *If $a_1, a_2 > 0$ the polynomial $q(x) = 1 + a_1 x - a_2 x^2$ is not type F.*

Proof. Suppose there exists $f(x) = \sum_{i=1}^{n} b_i x^i$ such that $(1 + f(x))q(x) = 1 - Q(x)$, i.e. such that

$$[x^k](1 + f(x))(1 + a_1 x - a_2 x^2) \leq 0$$

for $1 \leq k \leq n + 2$. Suppose this is possible for some definite $n = N > 1$. Then we will show it is possible for $n = N - 1$. By induction, this shows that this factorization should be possible for $n = 1$. However we will show that this it is impossible for $n = 1$, a contradiction.

Suppose first that $n = 1$. Then we have $(1 + f_1 x)(1 + a_1 x - a_2 x^2) = 1 + (a_1 + b_1)x + (a_1 b_1 - a_2)x^2 - a_2 b_1 x^3 \subseteq 1$. $-a_2 b_1 \leq 0$ implies $b_1 \geq 0$. We have $a_1 + b_1 \leq 0$ if and only if $b_1 \leq -a_1$. But then $0 \leq b_1 \leq -a_1 < 0$ a contradiction.

Now suppose $A(x) = (1 + a_1 x - a_2 x^2)(1 + \sum_{n=1}^{N} f_n x^n) \sqsubseteq 1$. We will show that also $B(x) = (1 + a_1 x - a_2 x^2)(1 + \sum_{n=1}^{N-1} f_n x^n) \sqsubseteq 1$.

Now $A(x) \sqsubseteq 1$ if and only if $[x^k]A(x) \leq 0$ for $1 \leq k \leq N-1$ and $[x^k]A(x) \leq 0$ for $k = N, N+1, N+2$. Also we have $B(x) \sqsubseteq 1$ if and only if $[x^k]B(x) \leq 0$ for $1 \leq k \leq N-1$ and $[x^k]B(x) \leq 0$ for $k = N, N+1$. For all $1 \leq k \leq N-1$ we have $[x^k]A(x) = [x^k]B(x)$. We'll show $[x^k]A(x) \leq 0$ for $k = N, N+1, N+2$ implies $[x^k]B(x) \leq 0$ for $k = N, N+1$ to complete the proof.

Solving the equations $[x^k]A(x) \leq 0$ for $k = N, N+1, N+2$ for b_n gives $0 \leq b_n \leq \min((a_2/a_1)b_{n-1}, a_2 b_{n-2} - a_1 b_{n-1})$. This gives $b_{n-1} \geq 0$ and $-a_2 b_{n-2} + a_1 b_{n-1} \leq 0$. These are equivalent to the equations $[x^k]B(x) \leq 0$ for $k = N, N+1$, namely $-a_2 b_{n-2} + a_1 b_{n-1} \leq 0$ and $-a_2 b_{n-1} \leq 0$. □

4.5 Partial Decision Procedures for the Skolem and Positivity Problems

We can now describe a partial decision procedure for the Positivity Problem. It will terminate with the correct output if the sequence has a negative term or if the characteristic polynomial is of Type F.

Algorithm 4.23. *Positivity Problem for Type F Characteristic Polynomials*

Input: A rational recurrence r_n with characteristic polynomial $q(x)$.

Output: The correct answer to the Positivity Problem if there exists n such that $r_n < 0$ or if $q(x)$ is Type F. Otherwise, it will not terminate.

1. Use the method of Theorem 4.7 to find $p(x)$ so that $\sum_{n \geq 0} r_n x^n = p(x)/q(x)$.
2. Set $\ell = 1$
3. Compute $r_{\ell-1}$ using the input initial terms and coefficients and previously computed r_j.
4. If $r_{\ell-1} < 0$, end and return "The recurrence is not positive, since $r_{\ell-1} < 0$."
5. Run step ℓ of Algorithm 4.20
6. If a witness $f(x), Q(x)$ was found in Step 5, run Algorithm 4.19 with $p(x), q(x), f(x), Q(x)$ to decide whether or not $p(x)/q(x) \sqsupseteq 0$, i.e. decide whether or not $r_n \geq 0$ for all n.
7. If no witness was found in Step 5, set $\ell \leftarrow \ell + 1$ and go to step 3.

Theorem 4.24. *Algorithm 4.23 operates as described and produces the correct output when it terminates.*

Proof. If there is an n such that $r_n < 0$, the algorithm will find it. Either it will find it directly in Step 3, or, according to Theorem 4.18, it will find it in Step 6.

If $r_n \geq 0$ and $q(x)$ is of Type F, then by Theorem 4.21, step 5 will eventually produce witnesses f and Q and then by Theorem 4.18, step 6 will produce the correct decision.

If $r_n \geq 0$ and $q(x)$ is not of Type F the algorithm will never terminate in step 3 or step 6 and hence will never terminate. □

We can use this algorithm to decide the Skolem Problem in some cases as well.

Algorithm 4.25. *Skolem Problem for Type F Characteristic Polynomials*

Input: A rational recurrence r_n with characteristic polynomial $q(x)$.

Output: The correct answer to the Skolem Problem if there exists n such that $r_n = 0$ or if the characteristic polynomial $q(x)$ of the recurrence $r_n^2 - 1$ is Type F. Otherwise it will not terminate.

1. Use the methods of Theorem 4.16 and Theorem 4.7 to find $p(x)$ and $q(x)$ so that $\sum_{n\geq 0} s_n x^n = p(x)/q(x)$ where $s_n = r_n^2 - 1$.
2. Use the method of Theorem 4.7 to get the initial terms and coefficients of the recurrence s_n from $p(x)$ and $q(x)$.
3. Set $\ell = 1$
4. Compute $s_{\ell-1}$ using the input initial terms and coefficients and previously computed s_j.
5. If $s_{\ell-1} < 0$, end and return "The recurrence has a zero term $r_{\ell-1} = 0$."
6. Run step ℓ of Algorithm 4.20
7. If a witness $f(x), Q(x)$ was found in Step 6, run Algorithm 4.19 with $p(x), q(x), f(x), Q(x)$ to decide whether or not $p(x)/q(x) \sqsupseteq 0$. If yes, $s_n \geq 0$ and $r_n \neq 0$ for all n. If no, then $s_n < 0$ and $r_n = 0$ for some n.
8. If no witness was found in Step 6, set $\ell \leftarrow \ell + 1$ and go to step 4.

Theorem 4.26. *Algorithm 4.23 operates as described and produces the correct output when it terminates.*

Proof. Since $s_n = r_n^2 - 1$, we have that $s_n < 0$ if and only if $r_n = 0$ and $p(x)/q(x) \sqsupseteq 0$ if and only if $r_n \neq 0$ for all n. The rest of the proof is as in the proof of Theorem 4.24. □

As we noted, the decidability of the Skolem Problem has been proven for orders less than or equal to 4 and the decidability of the Positivity Problems has been proven for orders less than or equal to 5. Our partial decision procedures for these problems can decide a great many particular cases with all higher orders, all recurrences whose characteristic polynomials are Type F.

Chapter 5: Directions for Future Research

A few questions naturally come to mind for further research. Here are some we plan to actively pursue.

The Bunk Bed Problem

1. A recent paper on the Bunk Bed Conjecture asserts that for every graph G there is an $\epsilon_G > 0$ such that if $p > 1 - \epsilon_G$ then the bunk bed conjecture is true for $BB(G, T)$ [HNNK21]. As they note, if there was an $\epsilon > 0$ such that $\epsilon_G \geq \epsilon > 0$ for *every* graph G, the Bunk Bed Conjecture would be true for some fixed p with $1 > p > 1 - \epsilon$ and hence, by our results, true in general. We plan to investigate the possibility of getting a such a uniform lower bound ϵ.

2. We discovered experimentally that for all bunk bed graphs $BB(G, T)$, the stronger statement $P(x_0 \leftrightarrow y_0) \geq P(x_0 \leftrightarrow^T y_0) \geq P(x_0 \leftrightarrow y_1)$ always held where $x_0 \leftrightarrow^T y_0$ is the event that x_0, y_0, and some vertex of T are in the same connected component. This is a conjecture not found in the literature. This conjecture, if true, would be sharper than the bunk bed conjecture, as we often have $P(x_0 \leftrightarrow y_0) > P(x_0 \leftrightarrow^T y_0)$. We plan to investigate this conjecture further.

Monotone Functions

1. In practice, we have found our Boolean function algorithms are greatly sped up by the following "tree trimming" trick. If a string $x \in T_n$ has been found to have $f(x) = 1$ then, since f is monotone, $f(y) = 1$ for every descendent y of x in T_n and T'_n. That means we don't need to evaluate f on any of those y and they can be *skipped*. We'd like to get some formal statement about how many computations of f this will save us in the average case.

2. It is of interest to know all the minimal subgraphs H of a graph G that satisfy some monotone property, such as $\chi(H) \geq 4$. Our algorithms can find these to give us experimental evidence for existing conjectures about such minimal examples.

The Skolem Problem

1. Can we characterize type F polynomials? Their roots seem to lie in a restricted area of the complex plane.

2. Can we extend our decision procedure Algorithm 4.19 for $p(x)/q(x) \sqsupseteq 0$ to a larger class of polynomials $q(x)$ than type F?

References

[BDJB10] Paul C. Bell, Jean-Charles Delvenne, Raphaël M. Jungers, and Vincent D. Blondel, *The continuous Skolem-Pisot problem*, Theoret. Comput. Sci. **411** (2010), no. 40-42, 3625–3634. MR 2724090

[BM76] Jean Berstel and Maurice Mignotte, *Deux propriétés décidables des suites récurrentes linéaires*, Bull. Soc. Math. France **104** (1976), no. 2, 175–184. MR 414475

[BP02] Vincent D. Blondel and Natacha Portier, *The presence of a zero in an integer linear recurrent sequence is NP-hard to decide*, vol. 351/352, 2002, Fourth special issue on linear systems and control, pp. 91–98. MR 1917474

[dB16] Paul de Buyer, *A proof of the bunkbed conjecture for the complete graph at* $p = \frac{1}{2}$, http://arxiv.org/abs/1604.08439 (2016).

[dBvETK51] N. G. de Bruijn, Ca. van Ebbenhorst Tengbergen, and D. Kruyswijk, *On the set of divisors of a number*, Nieuw Arch. Wiskunde (2) **23** (1951), 191–193. MR 0043115

[EH10] Herbert Edelsbrunner and John L. Harer, *Computational topology*, American Mathematical Society, Providence, RI, 2010, An introduction. MR 2572029

[EvdPSW03] Graham Everest, Alf van der Poorten, Igor Shparlinski, and Thomas Ward, *Recurrence sequences*, Mathematical Surveys and Monographs, vol. 104, American Mathematical Society, Providence, RI, 2003. MR 1990179

[GJ79] M. R. Garey and D. S. Johnson, *Computers and intractability: A guide to the theory of np-completeness (series of books in the mathematical sciences)*, first edition ed., W. H. Freeman, 1979.

[GK76] Curtis Greene and Daniel J. Kleitman, *Strong versions of Sperner's theorem*, J. Combinatorial Theory Ser. A **20** (1976), no. 1, 80–88. MR 389608

[Häg98] Olle Häggström, *On a conjecture of Bollobás and Brightwell concerning random walks on product graphs*, Combin. Probab. Comput. **7** (1998), no. 4, 397–401. MR 1680084 (2000i:60050)

[Häg03] ———, *Probability on bunkbed graphs*, Proceedings of FPSAC'03, Formal Power Series and Algebraic Combinatorics (2003).

[Han66] Georges Hansel, *Sur le nombre des fonctions booléennes monotones de n variables*, C. R. Acad. Sci. Paris Sér. A-B **262** (1966), A1088–A1090. MR 224395

[Han86] G. Hansel, *Une démonstration simple du théorème de Skolem-Mahler-Lech*, Theoret. Comput. Sci. **43** (1986), no. 1, 91–98. MR 847905

[HHH06] Vesa Halava, Tero Harju, and Mika Hirvensalo, *Positivity of second order linear recurrent sequences*, Discrete Appl. Math. **154** (2006), no. 3, 447–451. MR 2203195

[HNNK21] Tom Hutchcroft, Petar Nizić-Nikolac, and Alexander Kent, *The bunkbed conjecture holds in the $p \uparrow 1$ limit*, https://arxiv.org/abs/2110.00282 (2021).

[Jor10] Kelly Kross Jordan, *The necklace poset is a symmetric chain order*, J. Combin. Theory Ser. A **117** (2010), no. 6, 625–641. MR 2645181

[Kar90] Richard M. Karp, *The transitive closure of a random digraph*, Random Structures Algorithms **1** (1990), no. 1, 73–93. MR 1068492 (91j:05093)

[Knu11] Donald E. Knuth, *The art of computer programming. Vol. 4A. Combinatorial algorithms. Part 1*, Addison-Wesley, Upper Saddle River, NJ, 2011. MR 3444818

[Lec53] Christer Lech, *A note on recurring series*, Ark. Mat. **2** (1953), 417–421. MR 56634

[Lin11] Svante Linusson, *On percolation and the bunkbed conjecture*, Combin. Probab. Comput. **20** (2011), no. 1, 103–117. MR 2745680 (2012d:05355)

[Lip09] R.J. Lipton, *Mathematical embarrassments (blog post)*, https://rjlipton.wpcomstaging.com/2009/12/26/mathematical-embarrassments/ (December 2009).

[LT09] Vichian Laohakosol and Pinthira Tangsupphathawat, *Positivity of third order linear recurrence sequences*, Discrete Appl. Math. **157** (2009), no. 15, 3239–3248. MR 2554793

[Mah35] Kurt Mahler, *Einer arithmetische eigenschaft der taylor koeffizienten rationaler funktionen*, Proc. Akad. Wet. Amsterdam **38** (1935), 51–60.

[McD80] Colin McDiarmid, *Clutter percolation and random graphs*, Math. Programming Stud. (1980), no. 13, 17–25, Combinatorial optimization, II (Proc. Conf., Univ. East Anglia, Norwich, 1979). MR 592082 (81m:05119)

[MST84] M. Mignotte, T. N. Shorey, and R. Tijdeman, *The distance between terms of an algebraic recurrence sequence*, J. Reine Angew. Math. **349** (1984), 63–76. MR 743965

[OW12] Joël Ouaknine and James Worrell, *Decision problems for linear recurrence sequences*, Reachability problems, Lecture Notes in Comput. Sci., vol. 7550, Springer, Heidelberg, 2012, pp. 21–28. MR 3040104

[OW14] ———, *Positivity problems for low-order linear recurrence sequences*, Proceedings of the Twenty-Fifth Annual ACM-SIAM Symposium on Discrete Algorithms, ACM, New York, 2014, pp. 366–379. MR 3376387

[Pis02] Leonardo Pisano, *Fibonacci's liber abaci*, Sources and Studies in the History of Mathematics and Physical Sciences, Springer-Verlag, New York, 2002, A translation into modern English of Leonardo Pisano's ıt Book of calculation, Translated from the Latin and with an introduction, notes and bibliography by L. E. Sigler. MR 1923794

[RSA78] R. L. Rivest, A. Shamir, and L. Adleman, *A method for obtaining digital signatures and public-key cryptosystems*, Comm. ACM **21** (1978), no. 2, 120–126. MR 700103

[Sal76] Arto Salomaa, *Growth functions of Lindenmayer systems: some new approaches*, Automata, languages, development, 1976, pp. 271–282. MR 0502273

[Sin85] Parmanand Singh, *The so-called Fibonacci numbers in ancient and medieval India*, Historia Math. **12** (1985), no. 3, 229–244. MR 803579

[Sko35] Th. Skolem, *Ein verfahren zur behandlg gewisser exponentialer gleichungen*, 163–188.

[Soi76] M. Soittola, *On* D0L *synthesis problem*, Automata, languages, development, 1976, pp. 313–321. MR 0464748

[Sta12] Richard P. Stanley, *Enumerative combinatorics. Volume 1*, second ed., Cambridge Studies in Advanced Mathematics, vol. 49, Cambridge University Press, Cambridge, 2012. MR 2868112

[Tao07] Terrence Tao, *Open question: Effective skolem-mahler-lech theorem (blog post)*, https://terrytao.wordpress.com/2007/05/25/open-question-effective-skolem-mahler-lech-theorem/ (May 2007).

[vdBK01] J. van den Berg and J. Kahn, *A correlation inequality for connection events in percolation*, Ann. Probab. **29** (2001), no. 1, 123–126. MR 1825144

[Ver85] N.K. Vereshchagin, *The problem of appearance of a zero in a linear recurrence sequence (in russian)*, Mat. Zametki **38** (1985), no. 2.

[Yua] Qiaochu Yuan, https://math.stackexchange.com/questions/2520025/algorithm-for-computing-hadamard-product-of-two-rational-generating-functions.