2011

Three topics in the theory of computing: Multi-resolution cellular automata, the Kolmogorov complexity characterization of regular languages, and hidden variables in Bayesian networks

Brian Patterson
Iowa State University

Three topics in the theory of computing: Multi-resolution cellular automata, the Kolmogorov complexity characterization of regular languages, and hidden variables in Bayesian networks

by

Brian Patterson

A dissertation submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

Major: Computer Science

Program of Study Committee:
James I. Lathrop, Co-major Professor
Jack H. Lutz, Co-major Professor
Alicia L. Carriquiry
Vasant Honavar
Timothy McNicholl
Jin Tian

Iowa State University

Ames, Iowa

2011

DEDICATION

I would like to dedicate this thesis to my parents, Barb and Jim Patterson, without whose support I would not have been able to complete this work.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACKNOWLEDGEMENTS

I would like to take this opportunity to express my thanks to those who helped me with various aspects of conducting research and the writing of this thesis.

First and foremost, Drs. Jack H. Lutz and James I. Lathrop for their guidance, patience and support throughout this research and the writing of this thesis. Their insights and words of encouragement have often inspired me and renewed my hopes for completing my graduate education.

I would also like to thank my committee members for their efforts and contributions to this work: Drs. Alicia L. Carriquiry, Vasant Honavar, Timothy McNicholl, and Jin Tian. Dr. Giora Slutzki and Taylor Bergquist provided assistance without which the the work with regular languages and MRCA simulation would be woefully incomplete. I would also like to thank my Master's advisor, Dr. Dimitris Margaritis, for his assistance and expertise in my work with hidden variables in Bayesian networks.

CHAPTER 1. INTRODUCTION

It is a linchpin of scientific investigation that hypotheses are always subject to review. No matter how tight an experimental design, new evidence or a new perspective can render the common interpretation of any experiment false. A prominent example is the work of Albert Einstein in the development of general relativity [13]. Einstein explained space and time using a significantly different approach compared to that used in the previous 200 years since Isaac Newton. As a consequence, nuclear science became possible and extraordinary astronomical phenomena such as neutron stars, black holes, and gravitational waves were discovered [2]. However, the basic experimental results that Einstein based his theories on existed prior to those theories. He simply provided a new perspective.

Our work is centered around topics where we provide a new model or approach to a well-known paradigm. We provide a new lens through which to view an area of research, providing access for new researchers and perspectives. After a brief orientation with common terms in Chapter 2, we examine computation of real-valued sets in Chapter 3, our general multi-resolution cellular automata (MRCA) simulator in Chapter 4, how to prove languages are non-regular using Kolmogorov complexity in Chapter 5, and how to show hidden variables are valuable in Bayesian networks in Chapter 6. The remainder of this introduction is a summary of each of these areas and the main contributions of this dissertation to these areas.

1.1 Multi-Resolution Cellular Automata and Computable Analysis

In Chapter 3, we examine how the multi-resolution cellular automaton (MRCA) provides a new approach to understanding the field of computable analysis. This chapter is based on work with Lathrop and Lutz [28].

The primary objective of computability and complexity in analysis (or computable analysis for short) is to provide a realistic theoretical foundation for scientific computing. Specifically, large-scale, high-precision scientific computation requires this foundation in problem domains involving real numbers, functions on Euclidean spaces, differential equations, and other continuous mathematical objects.

The first task for computable analysis was to formulate notions of computability and complexity that are appropriate for such problem domains. It began with Turing [57; 58], who defined computable real numbers in 1936. It was furthered by Grzegorczyk [17] and Lacombe [27], who used the oracle Turing machine model from Turing's Ph.D. thesis [59] to define the computability of functions from real numbers to real numbers.

Progress accelerated dramatically in the 1980s. Pour-El, Richards, and Weihrauch conducted deep and influential investigations in computable analysis [46; 61], and Ko, Friedman, Kreitz, and Weihrauch formulated and investigated useful and informative models of the computational complexity of real-valued functions [22; 26; 23; 62]. This area is now a large and active research area that includes rigorous investigations of computation in Hilbert spaces, Banach spaces, differentiable manifolds, and many of the other mathematical settings of large-scale scientific computing.

Braverman and Cook [6] coined the term "bit-computability" for the approach shared by the above models and argued convincingly that it formed a good theoretical foundation for scientific computing. This is the definition we will use throughout this dissertation when referring to "computable" objects that include real values.

Our contribution is the introduction of a variation of the cellular automaton model for computable analysis. By reframing the questions of computable analysis in terms of cellular automata, we hope to interest newcomers in these questions as well as provide a new characterization of the complexities of these problems.

In the half century since their introduction by Ulam and von Neumann [7], cellular automata (CA) have been used in many ways. They have served as modeling tools for the physical, biological, and social sciences; been used to investigate speculative frontiers such as artificial life and hypercomputation; and shed new light on universality, parallelism, communication, fault tolerance, and other fundamental aspects of the theory of computing [7; 63; 16; 51; 32; 41]. Our primary interest is the last, specifically to use cellular automata to further our knowledge of the foundations of computable analysis.

Our initial focus is on the computability of subsets of Euclidean space, as defined by Brattka and Weihrauch [4; 62] and also exposited by Braverman [5; 6]. For ease of exposition and initial exploration, we confine ourselves to subsets of the 2-dimensional Euclidean unit square $[0,1]^2$. Informally, a subset X of $[0,1]^2$ is computable if there is an algorithm that turns pixels green when they are clearly in X and red when they are clearly not in X. A key feature of this definition is that the pixels have arbitrary, but finite, resolution.

More formally, for each rational point $q \in (\mathbb{Q}^2 \cap [0,1]^2)$ and each $n \in \mathbb{N}$, let $B(q, 2^{-n})$ denote the open ball of radius 2^{-n} centered at q (the set of points that are strictly less than distance 2^{-n} from a point q). A set $X \subseteq [0,1]^2$ is *bit computable* if there is a computable function $f : (\mathbb{Q}^2 \cap [0,1]^2) \times \mathbb{N} \to \{0,1\}$ such that, for all $q \in (\mathbb{Q} \cap [0,1])^2$ and $n \in \mathbb{N}$, the following two conditions hold.

(i) If $B(q, 2^{-n}) \subseteq X$, then $f(q,n) = 1$ (green).

(ii) If $B(q, 2^{1-n}) \cap X = \emptyset$, then $f(q,n) = 0$ (red).

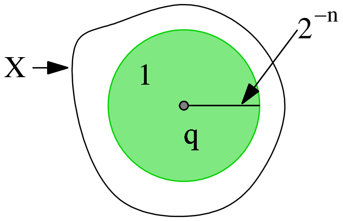


Figure 1.1 An example of a ball that is inside a set X so $f(q, n) = 1$.

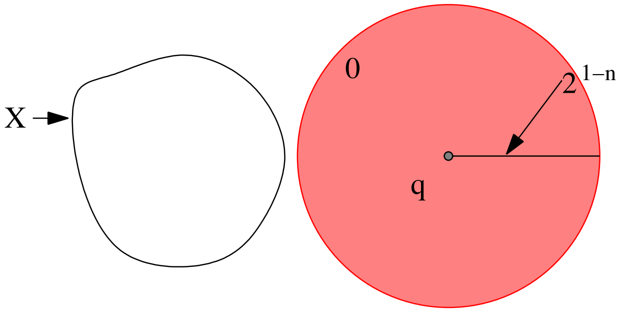


Figure 1.2 An example of a ball that is outside a set X so $f(q, n) = 0$.

If the hypotheses of (i) and (ii) are both false, then $f(q, n)$ must still be defined, and it may be either 1 or 0.

An example of (i) appears in Figure 1.1 and an example of (ii) appears in Figure 1.2. Figure 1.3 shows a couple of examples where neither (i) nor (ii) applies so the output of $f(q, n)$ is either 0 or 1. We need this "wiggle room" in defining sets of real coordinates to deal with the fact that most real numbers cannot be completely specified with a bounded amount of information.

We now introduce a cellular automaton model that achieves the spirit of this definition, with its cells corresponding to pixels on a computer screen. The first thing to note is that such a cellular automaton must allow arbitrary, but finite, precision. Accordingly, we allow cells to "fission" zero or more times during the course of execution. When a cell discovers that it is completely in or out of the set X, we want it to turn green or red (respectively) and then stay that color. Other cells may still be computing and/or fissioning, so we may

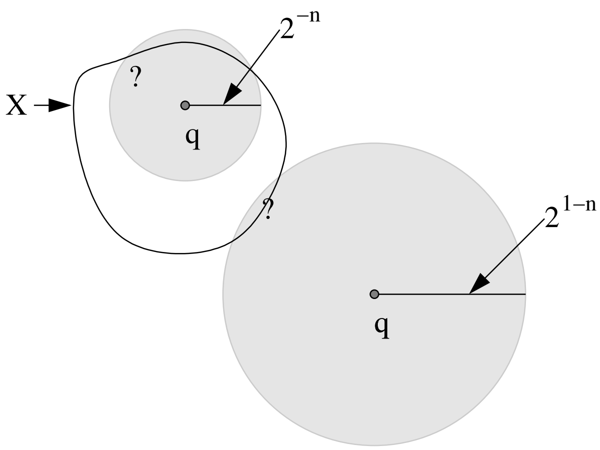


Figure 1.3 An example of some balls that are in neither case so $f(q, n)$ can be 0 or 1.

have cells of many different resolutions (sizes) at any given time.

In Section 3.1, our initial exploration of $[0, 1]^2$ using the tools of computable analysis yields that $X \subseteq [0, 1]^2$ is computable if and only if it is the separator of two computably open sets whose union is dense, with the caveat that X must have a computably nowhere dense boundary (Theorem 3.1.2). Informally, if a set X has a "thin" boundary, there are two computable functions to enumerate balls of points forming 2 sets covering $[0, 1]^2$ and X is the separator of these sets (i.e. X is a subset of one of the sets but not the other). We also investigate the possibility that the two sets could always be X° and $[0, 1]^2 \setminus X^\circ$ and find a counterexample that we call the Turing-gapped comb.

Section 3.2 introduces the multi-resolution cellular automaton (MRCA) which is a multidimensional cellular automaton that can, in addition to changing state depending on a small neighborhood, fission into multiple smaller cells, each half the size of the original in each dimension. We then define properties such as how a MRCA configuration is stored, how a neighbor is found, and what it means for an MRCA to compute a set of real points through colorings. Our main result in this area, presented in Section 3.3, is that a set X with computably nowhere dense boundary is computable if and only if it is MRCA computable (Theorem 3.3.2). This requires a complex construction outlined in this section and expanded in detail in Chapter 4.

Section 3.4 investigates constraining the model given in the previous section to polynomial time. We define polynomial time computable sets in the MRCA model as sets for which an approximation of the represented set to precision parameter t can be achieved in a number of steps polynomial in t. Our main result here is that a set X with a poly-time computably nowhere dense boundary is poly-time computable if and only if it is MRCA poly-time computable (Theorem 3.4.6). This requires a careful analysis of the construction used in the previous section.

1.2 An MRCA Simulator

Chapter 4 begins with Section 4.1, a primer on how to use our MRCA simulator. We write this simulator with the goal of attracting interest to the area. While Chapter 3 included several foundational reasons for which the MRCA model is useful in understanding real computation, this chapter focuses on how an MRCA can be used in a hands-on way to model computation as well as stoking the imagination of the user about how to use MRCAs to simulate other phenomena. We include a description and figures of how the simulator works as well as explain how rules specify transitions. Complete code for the simulator can be found in Appendix A. This chapter is based on work with Bergquist, Lathrop, and Lutz.

As outlined in the proof of Lemma 3.3.1 in Section 3.3, we can transform a one-dimensional CA for any computable cellular k-coloring ψ so it can be used to construct a k-coloring MRCA A such that, for each $i \in [k]$, $S_i(A) = S_i(\psi)$. Section 4.2 lays out the exact requirements for a set of transition rules for a one-dimensional CA that can be used for our construction and the subsequent transformations done to those rules to include in the specification of A.

Section 4.3 applies the ideas given in the previous two sections by explaining a set of one-dimensional CA rules for the set $L = \{(x, y) \in [0, 1]^2 \mid y < x^2\}$, what other rules must be specified regardless of CA, and how these rules interact. The construction is simplified by the fact that L is computable by a one-dimensional CA that always halts, so some of the alterations mentioned in the last section are not needed.

The final set of rules appears in Appendix B. Figures in this section show these rules in action, coloring L and $[0,1]^2 \setminus L$.

A more restricted and simple collection of subsets of $[0,1]^2$ consists of sets X that can be MRCA computed using MRCA rules that do not reference their neighbors (i.e., every cell transition is based entirely on the previous state of that cell). We show in Section 4.4 that MRCA rules written in this way (called *in-place MRCA rules*) can compute any region with rational lines as its boundaries (Theorem 4.4.3). Code for computing the rules for any rational line can be found in Appendix C. The classes in this appendix depend on the classes given in Appendix A and serve as an example of how MRCA-related classes can be used to perform automated rule construction.

1.3 Kolmogorov Complexity and Regular Languages

In Chapter 5, we reinterpret and expand results by Li and Vitányi [29] relating Kolmogorov complexity to regular languages. This chapter is based on work with Lathrop, Lutz, and Slutzki.

One of the main topics in a typical undergraduate computational theory course establishes a hierarchy of language families. The Chomsky Hierarchy singles out four classes of languages defined by the computational resources necessary to decide membership in the language: regular languages, context-free languages, context-sensitive languages, and recursively enumerable languages.

This chapter investigates a new way to show a language is nonregular, i.e., not a member language of the first class in the Chomsky Hierarchy. The Myhill-Nerode Theorem [19; 1] guarantees that a language L is regular if and only if it is the union of equivalence classes of a right-invariant equivalence relation of finite index on Σ^*. Using this theorem to directly show nonregularity is difficult to explain and unwieldy so a variety of alternative approaches exist. Most fall into the category of “pumping” lemmas as we review in Section 5.4. While pumping lemmas have been a part of the undergraduate theory curriculum for years, it is sometimes not intuitive to students and the version most easily proven is only a necessary but not a sufficient condition for regularity. This means that there exist languages that are not regular that still satisfy the common version of the pumping lemma (Theorem 5.4.4) and proving more complete versions of the pumping lemma requires advanced tools not covered in an undergraduate computational theory course [12].

We instead follow the approach of Li and Vitányi [29] to apply the Incompressibility Theorem of Kolmogorov complexity [24] (Theorem 5.1.4) to find a characterization of regular languages that makes our intuition (and students’ intuitions) about the link between (deterministic) finite state machines and regular languages rigorous and easy to apply.

In Section 5.1, after defining the Kolmogorov complexity $C(x)$ function on strings as the length of the shortest program to generate x, we show in Theorem 5.1.5 that there are infinitely many positive integers n for which at most 2^{c+1} strings $x \in \{0,1\}^n$ that satisfy $C(x) \leq c + \log n$ (Theorem 5.1.5). This improves the best-known bound from $2^{c+O(1)}$ [10] to 2^{c+1} using a simple method of proof more accessible to undergraduates.

Section 5.2 starts with Theorem 5.2.4, an undergraduate-accessible proof of how regular languages can be characterized by Kolmogorov complexity [29]. We also provide a new proof of the Kolmogorov complexity regularity lemma: language $A \subseteq \Sigma^*$ is regular if and only if there is a constant $d_A \in \mathbb{N}$ such that, for all $x, y_n^x \in \Sigma^*$, if y_n^x is the n^{th} string in A_x (counting from 0 in the standard ordering of Σ^*), then $C(y_n^x) \leq d_A + C(n)$ (Corollary 5.2.5). Our proofs are accessible to an undergraduate theory class and could be used in place of pumping lemma-based proofs of non-regularity.

We then show some example applications of the Kolmogorov complexity regularity lemma in Section 5.3 and explore prior work related to the pumping lemma in Section 5.4.

1.4 Essential Hidden Variables in Bayesian Networks

In Chapter 6, we investigate an alternative perspective on hidden variables in Bayesian networks. This leads to a different approach to hidden variable detection focused on those that improve the model. This chapter is based on work with Margaritis [43].

Since the introduction of Bayesian networks [44], the automated discovery and use of hidden variables (also called latent variables) to represent unmeasured or unmeasurable factors in a useful way has been an open problem. A Bayesian network specifies a joint probability distribution function (**pdf**) with a graph where each attribute of the data is represented by a vertex and paths in the graph indicate influence (or the absence of influence) between attributes. Rather than representing the joint pdf with a table containing the probability of each possible combination of attributes of interest, a collection of potentially smaller, local probability distributions of each attribute is used. Hidden variables are hypothesized attributes represented as nodes in the graph about which no experimental information is known.

It has been shown that Bayesian networks with hidden variables represent a larger class of probabilistic distributions than the class represented by Bayesian networks without hidden variables [14]. Supplementing this theoretical advance, the feasible discovery of hidden variables that enable this increased expressiveness would impact a variety of fields that utilize Bayesian networks.

Hidden variables may be used in Bayesian networks for semantic reasons [52] or for the compactness of the resulting Bayesian network [3]. * Most of the work in Artificial Intelligence (AI) has centered around the second aim—the use of hidden variables to simplify Bayesian networks while not altering the distribution that the network represents. However,

* Many in the field of artificial intelligence believe these aims are the same [45]. The central argument is that hidden variables that optimize a network must take advantage of some characteristic of the underlying probability distribution. Therefore, there must be something about the underlying distribution that allows the hidden variable to have an effect for a reasonable scoring method to find adding the hidden variable to be a good idea.

here we focus on detecting hidden variables that are essential to a more accurate representation of the underlying distribution.

Section 6.1 provides a motivating example and Section 6.2 provides background for the area of hidden variables in Bayesian networks. Section 6.3 provides an algorithm that supports our main result for this chapter: that the subset of the edge constraints given in the definition of W-networks (Definition 6.2.9) holds around all essential hidden variables. This means that the W-network is always found embedded in a network with a hidden variable and specifically that the hidden variable was always at the apex of the middle peak in the "W" of the W-network in the networks we examined. While these results may not generalize to networks of size larger than 8, this chapter provides a new perspective on the importance of investigating essential hidden variables separately from other types of hidden variables.

CHAPTER 2. PRELIMINARIES

We use the set $\mathbb{Z}$ of integers, the set $\mathbb{N}$ of natural numbers (i.e. nonnegative integers), the set $\mathbb{Q}$ of rational numbers, and the set $\mathbb{R}$ of real numbers.

$[0,1]$ represents a subset of $\mathbb{R}$ between 0 and 1 (inclusive) and $[0,1]^2$ represents the unit square of $\mathbb{R}^2$ with lower-left corner at the origin. The *open ball* with *center* $x \in [0,1]^2$ and *radius* $r \geq 0$ is the set

$$B(x,r) = \left\{ y \in [0,1]^2 \middle| |x-y| < r \right\},$$

where $|x-y|$ in this case denotes the Euclidean distance from x to y. We write $\mathcal{B}$ for the set of all such balls having rational centers ($x \in ([0,1] \cap \mathbb{Q})^2$) and rational radii ($r \in ([0,1] \cap \mathbb{Q})$). Note that $\mathcal{B}$ is a countable basis for the Euclidean topology on $[0,1]^2$.

For $X \subseteq [0,1]^2$, we use the standard topological notations X°, $\overline{X}$, and ∂X for the *interior*, *closure*, and *boundary* of X, respectively. For those not familiar, X° includes all points in X that can be the center of an open ball of any positive radius entirely contained in X. $\overline{X}$ include all points in X plus those points p for which, for every ball of finite radius r, $B(p,r)$ includes at least one point in X. ∂X is $\overline{X} \setminus X^\circ$ or equivalently $\overline{X} \cap ([0,1]^2 \smallsetminus X)$.

X° and any open ball is an example of an open set, a set where all points p are such that every ball of finite radius r, $B(p,r)$, includes only points in the set. $\overline{X}$ is an example of a closed set or set where all points p are such that every ball of finite radius r, $B(p,r)$, includes at least one other point in the set. Two sets in $[0,1]^2$ are *disjoint* if they have no points in common and two sets of sets in $[0,1]^2$ are *pairwise disjoint* if all pairs of sets in the set of sets are disjoint.

All logarithms in this dissertation are base 2. For each $n \in \mathbb{N}$, we write $[n] = \{0, 1, \ldots, n-1\}$. The function $[\![\ ]\!]$ maps boolean-valued propositions to $\{0,1\}$ — specifically, for a proposition ϕ, $[\![\phi]\!]$ is 1 if ϕ is true, 0 if it is not.

For each $n \in \mathbb{N}$ and $i, j \in [2^n]$, we define the *closed dyadic square*

$$Q(n,i,j) = \left[i \cdot 2^{-n}, (i+1) \cdot 2^{-n}\right] \times \left[j \cdot 2^{-n}, (j+1) \cdot 2^{-n}\right],$$

and we write $\mathcal{Q}$ for the set of all such squares. Note that $Q(0,0,0) = [0,1]^2$ and, as we shall see, $\mathcal{Q}$ is the set of all possible cells of an MRCA. We will always refer to i and j in binary, using subscript $_{binary}$ as a reminder of this fact where necessary.

A *language*, or *decision problem*, is a set $L \subseteq \{0,1\}^*$. We can identify a language L with its characteristic sequence χ_L defined by $\chi_L[n] = [\![s_n \in L]\!]$. As an example decision problem, recall the diagonal halting problem $K = \{n \in \mathbb{N} \mid \tau(n) < \infty\}$ where $\tau(n)$ is the running time of the n^{th} Turing machine on input n.

A language $L \in \Sigma^*$ is *computable* if there exists a Turing machine that takes a member of Σ^* as input and halts (i.e., terminates after a finite amount of time) with a correct indication of whether the input belongs in L or not. A language $L \in \Sigma^*$ is *computably enumerable* (c.e.) if there exists a Turing machine that takes a member of Σ^* as input and, if the input is in L, halts with a correct indication of this. Note that Turing machines witnessing that L is c.e. need not halt if the input is not in L. A language L is *co-c.e.* if there exists a Turing machine that takes a member of L as input and, if the input is not in L, halts with a correct indication of this. If L is c.e. and co-c.e., it is computable.

If a language is computable, we can find time bounds for the fastest Turing machine that computes it. Given a function $t : \mathbb{N} \to \mathbb{N}$, a language $L \in \Sigma^*$ is *t-time bounded* if there exists a Turing machine that takes a member of Σ^* of length n as input and halts after at most $t(n)$ steps with a correct indication of whether the input belongs in L. $L \in \Sigma^*$ is said to be *poly-time bounded* if $t(n)$ is a polynomial $p(n)$ (i.e., can be written in the form $\sum_{i=0}^{m} c_i n^i$ where $c_i \in \mathbb{Q}, m \in \mathbb{N}$). The set of all languages that are poly-time bounded is called P.

A *string* is any $w \in \{0,1\}^*$. We write $|w|$ for the length of string w and λ for the empty string. For $i, j \in \{0, \ldots, |w|-1\}$, we write $w[i \ldots j]$ for the string consisting of the i^{th} through the j^{th} bits of w and $w[i]$ for $w[i \ldots i]$, the i^{th} bit of w. Note that the 0^{th} bit $w[0]$ is the leftmost bit of w and that $w[i \ldots j] = \lambda$ if $i > j$.

A *sequence* is any $S \in \{0,1\}^\infty$. Notations $S[i \ldots j]$ and $S[i]$ are defined exactly as for strings. We work in the *Cantor space* $\mathbf{C}$ consisting of all sequences. A string $w \in \{0,1\}^*$ is a prefix of a sequence $S \in \mathbf{C}$, and we write $w \sqsubseteq S$, if the first $|w|$ bits of S is w.

Given an alphabet Σ, fix an ordering of Σ, and let $w_0^\Sigma, w_1^\Sigma, \ldots$ be the enumeration of Σ^*, first in order of length and, within each length, in the lexicographic order induced by the order on Σ. We will refer to this as the standard enumeration of Σ. When referring to the standard enumeration of $\{0,1\}^*$, we will use s_n for the n^{th} element where $s_0 = \lambda, s_1 = 0, s_2 = 1, s_3 = 00, s_4 = 01, \ldots$.

CHAPTER 3. MULTI-RESOLUTION CELLULAR AUTOMATA AND COMPUTABLE ANALYSIS

Our first and primary topic in this dissertation is the creation of a new model of computation, the multi-resolution cellular automata (MRCA), and its application to computable analysis. Although work so far has focused on showing the equivalence of MRCA computation to real computation (as defined in computable analysis), we hope that this new model will lead to new approaches to computable analysis and beyond.

3.1 Real Computation and Small Boundaries

In this section, we review some fundamental aspects of the computability of sets in the Euclidean plane and prove a new characterization of computable sets.

A set $G \subseteq [0,1]^2$ is *computably open* if there is a computably enumerable set $\mathcal{A} \subseteq \mathcal{B}$ such that $G = \cup\mathcal{A}$. Since each $B(x,r) \in \mathcal{B}$ is specified by its rational center and radius, it is clear what it means for a subset of $\mathcal{B}$ to be computably enumerable. This notion is easily seen to be incomparable with the notion of computability defined in the introduction. For example, if $\alpha \in (0,1)$ is a real number that is lower semicomputable but not computable, then $[0,\alpha)^2$ is computably open but not computable. Conversely, the square $\left[0,\frac{1}{2}\right]^2$ is computable but not open, hence not computably open.

A set $D \subseteq [0,1]^2$ is *dense* if it meets every nonempty open ball. A set $Z \subseteq [0,1]^2$ is *nowhere dense* if it is not dense in any nonempty open ball, i.e., for every $B \in \mathcal{B} \smallsetminus \{\emptyset\}$, there exists $B' \in \mathcal{B} \smallsetminus \{\emptyset\}$ such that $B' \subseteq B \smallsetminus Z$ (e.g., a line as in Figure 3.1). Effectivizing this,

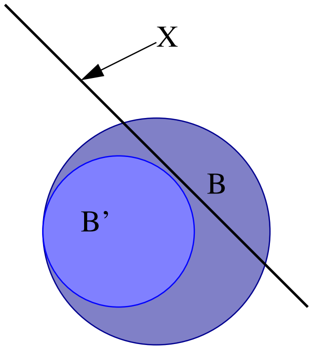


Figure 3.1 An example nowhere dense set X. Ball B' is evidence that X is nowhere dense with respect to B.

we say that Z is *computably nowhere dense* if there is a computable function $f : \mathcal{B} \smallsetminus \{\emptyset\} \to \mathcal{B} \smallsetminus \{\emptyset\}$ such that, for all $B \in \mathcal{B} \smallsetminus \{\emptyset\}$, $f(B) \subseteq B \smallsetminus Z$.

Nowhere dense sets are very small. For example, the Baire category theorem says that no countable union of nowhere dense sets contains all of $[0,1]^2$. Computably nowhere dense sets are very small in an even stronger sense [30].

Observation 3.1.1. *A set $Z \subseteq [0,1]^2$ is computably nowhere dense if and only if there is a computably open dense set $G \subseteq [0,1]^2$ such that $Z \cap G = \emptyset$.*

A *separator* of two (disjoint) sets A and B is a set S such that $A \subseteq S$ and $B \cap S = \emptyset$. The following result is crucial to our main theorem in Section 3.3.

Theorem 3.1.2. *If $X \subseteq [0,1]^2$ is a set whose boundary is computably nowhere dense, then the following two conditions are equivalent.*

(1) X is computable.

(2) X is a separator of two computably open sets whose union is dense.

Proof. Assume first that (2) holds. (We do not need the small-boundary hypothesis for this direction of the proof.) Then we have computably open sets $G, H \subseteq [0,1]^2$ such that X is a separator of G and H and $G \cup H$ is dense on $[0,1]^2$. Since G and H are computably open, there exist c.e. sets $\mathcal{A}_G$, $\mathcal{A}_H \subseteq \mathcal{B}$ such that $G = \bigcup \mathcal{A}_G$ and $H = \bigcup \mathcal{A}_H$. Let $f : (Q^2 \cap [0,1]^2) \times \mathbb{N} \to \{0,1\}$ be computed by Algorithm 1.

```
Algorithm 1  $f(q,r) = y$
             Input: $(q,r) \in (Q \cap [0,1])^2 \times \mathbb{N}$.
             Output: $y \in \{0,1\}$.

Find $B \in \mathcal{B}$ such that
(i) $B \in \mathcal{A}_G$ and $B \cap B(q, 2^{1-r}) = \emptyset$
or
(ii) $B \in \mathcal{A}_H$ and $B \cap B(q, 2^{-r}) \neq \emptyset$
if (i) holds then
    return 1
else
    return 0
end if
```

Since $G \cup H$ is dense, any ball $B(q, 2^{-r})$ must meet $G \cup H$. If $B(q, 2^{-r})$ meets H then a set B as in (ii) exists. If not, then $B(q, 2^{-r})$ meets G, whence a set B as in (i) exists. Since $\mathcal{A}_G$ and $\mathcal{A}_H$ are c.e., it follows that a set B satisfying (i) or (ii) can be found. Hence our algorithm always halts, and f is a computable, total function. Now

$$\begin{aligned} B(q, 2^{-r}) \subseteq X \quad &\Rightarrow \quad B(q, 2^{-r}) \cap H \neq \emptyset \\ &\Rightarrow \quad f(q,r) = 1, \end{aligned}$$

and

$$\begin{aligned} B(q, 2^{1-r}) \cap X = \emptyset \quad &\Rightarrow \quad B(q, 2^{1-r}) \cap G = \emptyset \\ &\Rightarrow \quad f(q,r) = 0, \end{aligned}$$

so f testifies that X is computable, i.e., (1) holds.

Conversely, assume that (1) holds and that ∂X is computably nowhere dense. By Observation 3.1.1 there is a computably open dense set $D \subseteq [0,1]^2$ such that $D \cap \partial X = \emptyset$. Since D is computably open, there is a c.e. set of balls $\mathcal{D} \subseteq \mathcal{B}$ such that $D = \bigcup \mathcal{D}$. Since $D \cap \partial X = \emptyset$

and $\{X^\circ, ([0,1]^2 \smallsetminus X)^\circ, \partial X\}$ is a partition of $[0,1]^2$, it must be the case that every ball $B \in \mathcal{D}$ satisfies $B \subseteq X^\circ$ or $B \subseteq ([0,1]^2 \smallsetminus X)^\circ$. Otherwise, $(B \cap X^\circ, B \cap ([0,1]^2 \smallsetminus X)^\circ)$ would form a disconnection of B, contradicting the connectedness of balls. We now show that the set

$$\mathcal{D}_G = \{B \in \mathcal{D} \mid B \subseteq X^\circ\}$$

is c.e. Let $f : (\mathbb{Q} \cap [0,1])^2 \times \mathbb{N} \to \{0,1\}$ testify to the computability of X and $B = B(q,r) \in \mathcal{D}$. Note three things.

(i) $r = 0 \Rightarrow B \in \mathcal{D}_G$.

(ii) If $r \neq 0$, let $n(r)$ be the least positive integer such that $2^{-n(r)} < r$. Then

$$\begin{aligned} f(q, n(r)+1) = 1 \quad &\Rightarrow \quad f(q, n(r)+1) \neq 0 \\ &\Rightarrow \quad B(q, 2^{-n(r)}) \cap X \neq \emptyset \\ &\Rightarrow \quad B \cap X \neq \emptyset \\ &\Rightarrow \quad B \cap X^\circ \neq \emptyset \\ &\Rightarrow \quad B \subseteq X^\circ \\ &\Rightarrow \quad B \in \mathcal{D}_G. \end{aligned}$$

(iii) If $r \neq 0$, then

$$\begin{aligned} f(q, n(r)+1) = 0 \quad &\Rightarrow \quad f(q, n(r)+1) \neq 1 \\ &\Rightarrow \quad B(q, 2^{-n(r)+1}) \not\subseteq X \\ &\Rightarrow \quad B \not\subseteq X \\ &\Rightarrow \quad B \notin \mathcal{D}_G. \end{aligned}$$

Taken together, these three things show that

$$\mathcal{D}_G = \{B(q,r) \in \mathcal{D} \mid r = 0 \text{ or } f(q, n(r)+1) = 1\}.$$

Since $\mathcal{D}$ is c.e. and f is computable, it follows that $\mathcal{D}_G$ is c.e.

A similar argument shows that the set

$$\mathcal{D}_H = \left\{ B \in \mathcal{D} \mid B \subseteq ([0,1]^2 \smallsetminus X)^\circ \right\}$$

is c.e.

Let $G = \bigcup \mathcal{D}_G, H = \bigcup \mathcal{D}_H$. Then G and H are computably open. Also, $G \subseteq X^\circ$ and $H \subseteq ([0,1]^2 \smallsetminus X)^\circ$, so X is a separator of G and H. Finally, $G \cup H = D$ is dense in $[0,1]^2$, so (2) holds. □

The small-boundary hypothesis is needed here. For example, if $X = \left[0, \frac{1}{2}\right]^2 \cup (\mathbb{Q} \cap [0,1])^2$, then (1) holds, but (2) fails.

It is tempting to think that the two computably open sets in (2) can always be the interior of X and its complement. We now give an example of a computable set X with computably nowhere dense boundary whose interior is not computably open.

Let

$$X = [0,1]^2 \smallsetminus \bigcup_{n=2}^{\infty} G_n,$$

where

$$G_n = \left(\frac{1}{n} - \epsilon(n), \frac{1}{n} + \epsilon(n) \right) \times \left(0, \frac{1}{2} \right),$$

$$\epsilon(n) = 2^{-(\tau(n)+n+2)},$$

and $\tau(n)$ is the running time of the n^{th} Turing machine on input n. Note that $\epsilon(n) = 0$ and $G_n = \emptyset$ if $\tau(n) = \infty$.

Since each "gap" G_n is open, the set X is closed. Since X has gaps whose width is dictated by the run time of Turing machines, we call X the *Turing-gapped comb* (shown in Figure 3.2).

Lemma 3.1.3. *For all $m, n \in \mathbb{N}$, G_m and G_n are pairwise disjoint.*

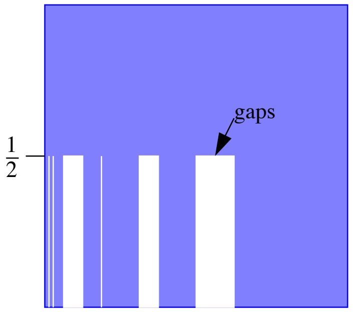


Figure 3.2 The Turing-gapped comb (appearing in blue).

Proof. Let G_n and G_{n+1} be adjacent gaps. Let ℓ_n be the x-coordinate of the left border of G_n, and let r_{n+1} be the x-coordinate of the right border of G_{n+1}. Then

$$\begin{aligned}
\ell_n - r_{n+1} &= \left(\frac{1}{n} - \epsilon(n)\right) - \left(\frac{1}{n+1} + \epsilon(n+1)\right) \\
&= \left(\frac{1}{n} - \frac{1}{n+1}\right) - (\epsilon(n) + \epsilon(n+1)) \\
&= \frac{1}{n(n+1)} - (\epsilon(n) + \epsilon(n+1)) \\
&\geq \frac{1}{n(n+1)} - \left(2^{-(n+2)} + 2^{-(n+3)}\right) \\
&= \frac{1}{n(n+1)} - \frac{3}{2^{n+3}} \\
&= \frac{2^{n+3} - 3n(n+1)}{2^{n+3} \cdot n(n+1)} \\
&> 0,
\end{aligned}$$

as $n \geq 2$. □

Lemma 3.1.4. *The Turing-gapped comb set X is computable.*

Proof. For each $k \in \mathbb{N}$, define the "fat set"

$$F_k = \left\{ n \in \mathbb{N} \mid \epsilon(n) \geq 2^{1-k} \right\}.$$

It is easy to check that

$$F_k = \{n \in \mathbb{N} \mid \tau(n) \leq k - n - 3\}.$$

Note that each F_k is finite, and there is an algorithm that, given $k \in \mathbb{N}$, lists the elements of F_k.

Define $f : (\mathbb{Q}^2 \cap [0,1]^2) \times \mathbb{N} \to \{0,1\}$ as

$$f(q,k) = \begin{cases} 0 & \text{if } (\exists n \in F_k)\ B(q, 2^{1-k}) \subseteq G_n \\ 1 & \text{otherwise.} \end{cases}$$

It is clear that f is computable. Also,

$$\begin{aligned} B(q, 2^k) \subseteq X \quad &\Rightarrow \quad (\forall n) B(q, 2^{-k}) \cap G_n = \emptyset \\ &\Rightarrow \quad (\forall n) B(q, 2^{1-k}) \not\subseteq G_n \\ &\Rightarrow \quad f(q,k) = 1, \end{aligned}$$

and

$$\begin{aligned} B(q, 2^{1-k}) \cap X = \emptyset \quad &\Rightarrow \quad (\exists n) B(q, 2^{1-k}) \subseteq G_n \\ &\Rightarrow \quad (\exists n \in F_k) B(q, 2^{1-k}) \subseteq G_n \\ &\Rightarrow \quad f(g,k) = 0, \end{aligned}$$

so f testifies that X is computable. □

Lemma 3.1.5. *If $f : \mathbb{N} \to \mathcal{B}$ satisfies*

$$\bigcup_{j=0}^{\infty} f(j) = X^\circ,$$

where X is the Turing-gapped comb then the diagonal halting problem K is co-c.e. relative to f.

Proof. Assume the hypothesis. Then for all $n \geq 2$,

$$\begin{aligned} n \in K &\Leftrightarrow \left(\frac{1}{n}, \frac{1}{4}\right) \notin X^\circ \\ &\Leftrightarrow (\forall j \in \mathbb{N}) \left(\frac{1}{n}, \frac{1}{4}\right) \notin f(j). \end{aligned}$$

So, to find whether the n^{th} Turing machine halts, we enumerate balls $f(j)$ of X° until $\left(\frac{1}{n}, \frac{1}{4}\right) \in f(j)$. This process will halt if and only if $n \notin K$ so K is co-c.e. relative to f (given our hypothesis). □

Corollary 3.1.6. *The interior of the Turing-gapped comb X° is not computably open.*

Proof. Since K is not co-c.e. relative to any computable function, Lemma 3.1.5 tells us that there is no computable $f : \mathbb{N} \to \mathcal{B}$ such that $\bigcup_{j=0}^{\infty} f(j) = X^\circ$. Hence X° is not computably open. □

Thus, by Lemma 3.1.4 and Corollary 3.1.6, the Turing-gapped comb is computable but its interior is not computably open. We cannot always use the interior of a set and its complement as the two sets that a set separates in part (2) of Theorem 3.1.2.

It is interesting to note that the Turing-gapped comb is also regular, meaning that $\overline{X^\circ} = X$. Taking the interior of the set simply removes its border and taking the closure restores only and exactly that border (this is why it is key that the gaps be open sets). This implies that the comb is not a “bizarre” set in a topological sense.

3.2 Introduction to MRCAs

We now introduce the multi-resolution cellular automaton (MRCA), a model of computation that generalizes cellular automata by allowing cells to fission.

Let $U_2 = \{(0,1), (1,0), (0,-1), (-1,0)\}$ be the set of direction vectors and let S be a finite set of states. We write $S_\perp = S \cup \{\perp\}$, where $\perp$ is an “undefined” symbol. We define a *neighborhood status* for S to be a function $\nu : U_2 \times \{0,1\} \to (S_\perp)^2$*, and we write N_S for the set of all such ν.

Definition. A *multi-resolution cellular automaton (MRCA)* is a triple

$$A = (S, \delta, s)$$

where S is a finite set of states; $s \in S$ is the *start state*; and

$$\delta : S \times N_S \to S \cup S^4$$

is the *transition function*.

In order to specify the operation of an MRCA, we need a careful understanding of the neighborhood of a cell Q. For each $Q \in \mathcal{Q}$, $\mathbf{u} \in U_2$, and $b \in \{0,1\}$, let $Q_{\mathbf{u},b}$ be the dyadic closed square defined as follows.

(i) $Q_{\mathbf{u},b}$ is adjacent to Q on side $\mathbf{u}$ of Q.

(ii) The side-length of $Q_{\mathbf{u},b}$ is half that of Q.

(iii) If $\mathbf{u} = \pm(0,1)$, then $Q_{\mathbf{u},0}$ lies below $Q_{\mathbf{u},1}$; if $\mathbf{u} = \pm(1,0)$, then $Q_{\mathbf{u},0}$ lies to the left of $Q_{\mathbf{u},1}$.

Note that $Q_{\mathbf{u},b} \in \mathcal{Q}$, unless Q abuts $[0,1]^2$ to direction $\mathbf{u}$.

A *multi-resolution* is a finite set $\mathcal{R} \subseteq \mathcal{Q}$ with the following two properties.

(i) For all $Q_1, Q_2 \in \mathcal{R}$, $Q_1 \neq Q_2 \Rightarrow Q_1^\circ \cap Q_2^\circ = \emptyset$.

(ii) $\bigcup \mathcal{R} = [0,1]^2$.

That is, $\mathcal{R}$ is a finite set of cells that do not overlap (except perhaps along their edges) and that cover $[0,1]^2$.

*For reasons to be discussed soon, the neighbor to a direction from our current cell includes up to two cells of one size smaller.

A *configuration* of an MRCA $A = (S, \delta, s)$ is an ordered pair $\gamma = (\mathcal{R}, \sigma)$, where $\mathcal{R}$ is a multi-resolution and $\sigma : \mathcal{R} \to S$ is an assignment of states to cells in $\mathcal{R}$. We write $CONFIG_A$ for the set of all configurations of A.

If $\gamma = (\mathcal{R}, \sigma)$ is a configuration of an MRCA $A = (S, \delta, s)$ and $Q \in \mathcal{R}$, then the *neighborhood status of Q in γ* is the function $\nu_{\gamma,Q} \in N_S$ defined by

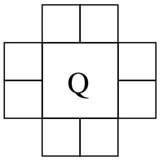
$$\nu_{\gamma,Q}(\mathbf{u}, b) = \begin{cases} \sigma(Q') & \text{if } Q_{\mathbf{u},b} \subseteq Q' \in \mathcal{R}; \\ \perp & \text{if no such } Q' \text{ exists.} \end{cases}$$

Note that $\nu_{\gamma,Q}(\mathbf{u}, b) = \perp$ if either

(i) Q abuts side $\mathbf{u}$ of $[0, 1]^2$, or

(ii) $\mathcal{R}$ uses two or more cells smaller than $Q_{\mathbf{u},b}$ to cover $Q_{\mathbf{u},b}$.

Intuitively, the "neighbors" of a cell Q are the eight surrounding cells suggested by the following picture.



If any of the neighboring cells are outside of $[0, 1]^2$ or have already subdivided, their states are undefined ($\perp$) for the purpose of updating Q. If some existing cell contains two of these neighboring cells, its state is used for both $b = 0$ and $b = 1$ neighbor. If the value of δ, applied to this neighborhood information, is in S, then this is the new state of Q.

We can view an MRCA configuration in a number of different ways. The most common and the one we use for most of this dissertation is depicted in Figure 3.3(b). However, when configuration information is stored, the data structure used to store location and state is more accurately depicted as Figure 3.3(a) — a quadrary tree where each branch indicates a fission to a set direction and the leaves are cells. For any fission of a cell c in this view, let

$00, 10, 01$, and 11 be the string representation of the location of the southwest, northwest, southeast, and northeast child cells of c. So, for example, there is no actual cell n_2 in Figure 3.3(a) because it has children (indicating fission). It has cells with state 1 and 0 in its southeast and northeast quadrants while its other two children have fissioned.

A final representation of the location information is as a prefix-free set $\mathcal{R}$ and a function mapping those strings to symbols σ as appearing in the caption of Figure 3.3. For example, the cell marked x can be written as at location 1001, the concatanation of its ancestor's location strings. This last interpretation allows us to more succinctly discuss the complexity of a configuration using tools like Kolomogorov complexity.

We now examine the computation of $\nu_{\gamma,Q}(\mathbf{u}, b)$ using a few simple, computable functions. $set : U_2 \rightarrow \{0,1\}^2 \times \{0,1\}^2$ is used to discover the location string for fissioned cells to a particular direction and is defined as $set((0,1)) = \{01, 11\}$, $set((0,-1)) = \{00, 10\}$, $set((1,0)) = \{11, 10\}$, and $set((-1,0)) = \{00, 01\}$. $\oplus : \{0,1\}^2 \times U_2 \rightarrow \{0,1\}^2$ is defined as: $\forall j \in \{0,1\}\ 0j \oplus (1,0) = 1j$, $1j \oplus (-1,0) = 0j$, $j0 \oplus (0,1) = j1$, and $j1 \oplus (0,-1) = j0$. The $x \oplus \mathbf{u}$ operation will be referred to as *flipping x in direction $\mathbf{u}$* and is used to specify the location of a neighbor cell to a direction (either vertically or horizontally).

Using these functions, $\nu_{\gamma,Q}$ is given for two dimensions in Algorithms 2 and 3. A Java implementation appears in Appendix A in the class `Configuration2D` as the method named `neighCell`. Higher-dimensional versions of $\nu_{\gamma,Q}$ can be derived similarly.

Example 3.2.1. See Figure 3.3, where we are seeking the north neighbor of x (which is call y).

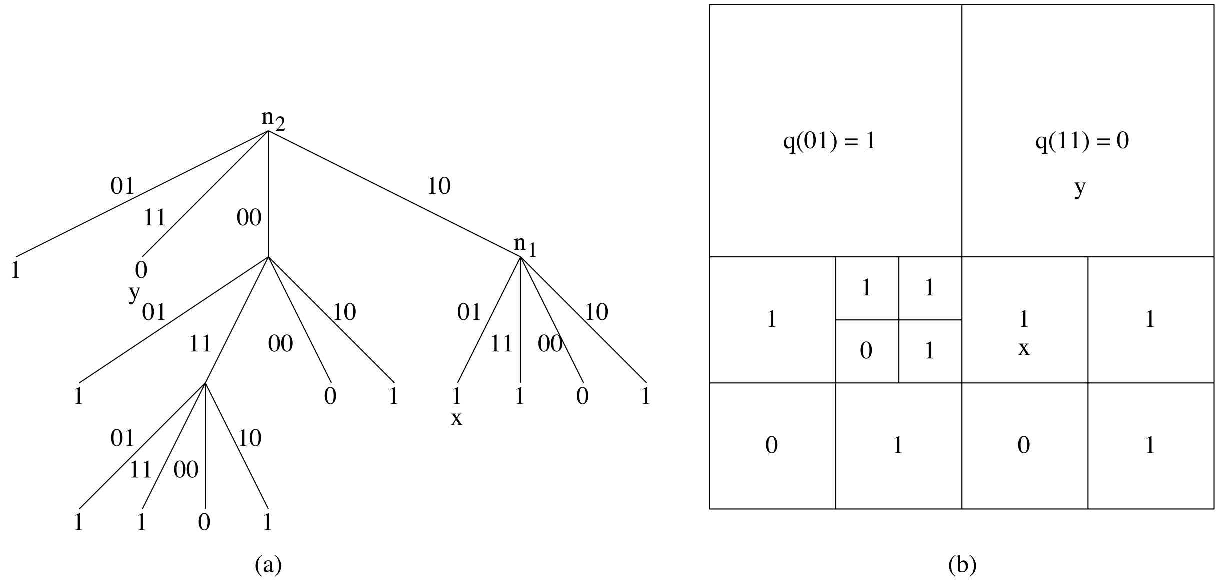
1. Trace from the leaf node corresponding to x up until we find an ancestor node that is not one of the nodes in direction $\mathbf{u}$ (ignoring the other part of the direction). For example, a node seeking its north parent ($\mathbf{u} = (0,1)$) traces its lineage until it reaches a node n_1 for whom n_1 is not a north child of its parent. This accomplishes a "zooming out" to the fission that placed x and its ancestors to the south of n_1. If such a node

Algorithm 2 $\sigma(y) = \nu_{\gamma,Q}(\mathbf{u}, b)$
Input: $\gamma \in CONFIG_A$; $Q \in \mathcal{R}$; $\mathbf{u} \in U_2$; $b \in [0,1]$.
Output: $\sigma(y)$.

```
$cur \leftarrow neighRec(\gamma, Q, \mathbf{u}, b)$
if $cur$ is $null$ or $cur$'s children have children then
    return $\perp$
else
    if $cur$ has no children then
        return $\sigma(cur)$
    else
        $cur \leftarrow b$ cell in direction $\mathbf{u}$ from $Q$
        return $\sigma(cur)$
    end if
end if
```

Algorithm 3 $y = neighRec(\gamma, Q, \mathbf{u}, b)$
Input: $\gamma \in CONFIG_A$; $Q \in \mathcal{R}$; $\mathbf{u} \in U_2$; $b \in [0,1]$.
Output: $y \in \gamma$.

```
$cur \leftarrow Q$
if $cur.parent == null$ then
    return $null$
else if $\exists \ell \in set(\mathbf{u})$ $cur.parent.child[\ell] == cur$ then
    $cur \leftarrow neighRec(\gamma, cur.parent, \mathbf{u}, b)$
    $\ell \leftarrow \ell \oplus d$
    if $cur$ has children then
        $cur \leftarrow cur.child[\ell]$
    end if
else
    Let $\ell \in U_2$ such that $cur.parent.child[\ell] == cur$
    $cur \leftarrow cur.parent$
    $\ell \leftarrow \ell \oplus d$
    $cur \leftarrow cur.child[\ell]$
end if
return $cur$
```



(a)

(b)

Figure 3.3 Tree and physical representation of $\gamma = (\mathcal{R}, \sigma)$ with $\mathcal{R} = \{01, 11, 0001, 0000, 0010, 001101, 001111, 001100, 001110, 1001, 1011, 1000, 1001\}$ and $\sigma = \{(01, 1), (11, 0), (0001, 1), (0000, 0), (0010, 1), (001101, 1), (001111, 1), (001100, 0), (001110, 1), (1001, 1), (1011, 1), (1000, 0), (1001, 1)\}$. The cell marked by x denotes an example cell we find the north neighbor of, y. n_i denotes internal, data structure tree nodes encountered in Example 3.2.1.

cannot be found (i.e. we trace back to the root of the tree), there is no neighbor to the north so return $\perp$.

2. Traverse to the parent of this node (n_2 in Figure 3.3). Again, if such a parent does not exist, return $\perp$.

3. "Flipped" direction ℓ is identified and we go to the sibling in that direction. In our example, we want the sibling cell directly to the north of n_1, y.

4. Traverse to the child of this sibling in the "flipped" direction from the direction we took to reach this point until we run out of children. We trace the edges used to reach n_1 but flip the north/south direction. Intuitively, we're following fissions as deep as we can into the cell north of our original cell x by always going to the southeast child. In our example, we are already at a most-fissioned element of γ in the subtree rooted at y.

5. If the result y has no children, we return y. Otherwise y represents a cell that has fissioned more than x by this configuration. If y's children's children exist to the flip direction, we return $\perp$ as the cell has fissioned too much to be read in our model. Else, we return the child of y corresponding to our input $b \in \{0, 1\}$.

We now prove the correctness of Algorithm 2.

Lemma 3.2.2. *Given an MRCA cell Q, direction $\mathbf{u}$, configuration γ, and child index b, Algorithm 2 correctly returns $\nu_{\gamma,Q}(\mathbf{u}, b)$ as the neighbor state to direction $\mathbf{u}$ (cell b) of x.*

Proof. Let x's neighbor to direction $\mathbf{u}$ be $x_{\mathbf{u}}$. There are four classes of cases to consider when finding $x_{\mathbf{u}}$.

Case 1. x has no neighbor as it is at the edge of $[0, 1]^2$. We have defined the neighbor as having state $\perp$ in this case. If x has no neighbor in direction $\mathbf{u}$ then it is to the direction $\mathbf{u}$ in any fission. Thus we will always be able to find an $i \in \mathit{set}(\mathbf{u})$ until we reach the root, at which point Algorithm 3 finds that $x.\mathit{parent}$ is null and consequently Algorithm 2 returns $\perp$.

Case 2. x has neighbors that are the result of 2 or more fissions than x. $\nu_{\gamma,Q}(\mathbf{u}, b)$ defines the neighbor as having state $\perp$. This case is handled by explicitly testing in Algorithm 2 if the returned value *cur* must still fission two or more times to reach an actual cell after fissioning an equal number of times from the shared ancestor with x. If this is true, Algorithm 2 returns $\perp$. Thus this case is treated as case 4 until the final test for being a leaf in the tree representation of γ and returns the correct value in this case.

Case 3. By the same reasoning as in case 2, if x has neighbors that are the result of exactly one fission more than x, we correctly post-process the return value from Algorithm 3 to select the correct neighbor cell using index b.

Case 4. x has neighbors that are the result of the same or fewer fissions than x. We will show that $x_{\mathbf{u}}$ is found to be a neighbor in Algorithm 2 by induction on the number of recursions done by Algorithm 3.

The base case is that $x_{\mathbf{u}}$ and x are siblings in γ. There is no recursion to find $x_{\mathbf{u}}$ and the last conditional block of Algorithm 3 is executed. There will not be an $\ell \in \mathit{set}(\mathbf{u})$ such that $\mathit{cur}.\mathit{parent}.\mathit{child}[\ell] == \mathit{cur}$ because $x_{\mathbf{u}}$ is in direction d from x at the current fission level and *cur* cannot be in direction d. Thus, according to Algorithm 3, *cur* becomes the shared parent of x and $x_{\mathbf{u}}$. The next step flips ℓ by $\mathbf{u}$ and sets *cur* to the child $x_{\mathbf{u}}$. The flip takes the fission direction of x and changes only the direction of concern. $x_{\mathbf{u}}$ exists and has no children so we correctly return $q(x_{\mathbf{u}})$ from Algorithm 2.

We now examine the inductive case where $x_{\mathbf{u}}$ and x are not siblings but are nonetheless neighbors. Assume through induction that calls to find the neighbor of x's parent will return an ancestor of $x_{\mathbf{u}}$ as it must be the neighbor to direction $\mathbf{u}$ of x's parent.

By the principle of induction, x's parent is not null since $x_{\mathbf{u}}$ exists and therefore shares an ancestor with x. Note that x is a child to direction $\mathbf{u}$ in its parent's fission else we would enter the base case. By the inductive hypothesis, the recursive call $\mathit{neighRec}(\gamma, \mathit{cur.parent}, \mathbf{u}, b)$ sets cur so it refers to a cell that has $x_{\mathbf{u}}$ in it's subtree. Call this cell $p(x_{\mathbf{u}})$. If $x_{\mathbf{u}}$ were not in $p(x_{\mathbf{u}})$'s subtree, it would not be to direction $\mathbf{u}$ from x's parent and would not be adjacent to x in the physical interpretation.

After the recursion is complete, cur is set to the ℓ child of $p(x_{\mathbf{u}})$. Thus, when we flip ℓ by $\mathbf{u}$, we take cur to be the child of $p(x_{\mathbf{u}})$ in the direction closest to x if $p(x_{\mathbf{u}})$ fissioned. Otherwise there is no change in cur. We now move cur to the child in direction ℓ of $p(x_{\mathbf{u}})$ which is $x_{\mathbf{u}}$. So Algorithm 3 returns $x_{\mathbf{u}}$.

Finally, we verify Algorithm 2 correctly returns $\sigma(x_{\mathbf{u}})$ given Algorithm 3 is correct. Clearly $\mathit{cur} \neq$ null. By the principle of induction, cur represents a node of depth at most equal to x in C. Since we assumed for this case that $x_{\mathbf{u}}$ is larger than or equal in size to x, we correctly return $\sigma(x_{\mathbf{u}}) = \sigma(\mathit{cur})$.

□

We now discuss MRCA transition functions and configurations. If the value of transition function δ is a single value in S, Q does not fission and changes to that state. If the value of transition function δ is $(s_1, s_2, s_3, s_4) \in S^4$, then Q fissions into the four cells

Q_2	Q_1
Q_3	Q_4

and each cell Q_i is assigned a state s_i. The *extended transition function* of an MRCA $A = (S, \delta, s)$ is the function

$$\delta^* : \mathit{CONFIG}_A \to \mathit{CONFIG}_A$$

defined as follows. For each $\gamma = (\mathcal{R}, \sigma) \in \mathit{CONFIG}_A$, we set $\delta^*(\gamma) = (\mathcal{R}', \sigma')$, where

(i) $\mathcal{R}'$ is obtained from $\mathcal{R}$ by replacing each cell Q for which $\delta(\nu_{\gamma,Q}) = (s_1, s_2, s_3, s_4) \in S^4$ with the four cells Q_1,Q_2,Q_3, and Q_4 that are the upper right, upper left, lower left, and lower right quadrants, respectively, of Q; and

(ii) σ' is obtained from σ by setting each $\sigma'(Q_i) = s_i$ when Q_1,Q_2,Q_3,Q_4 are as in case (i) and setting $\sigma'(Q_i) = \delta(\nu_{\gamma,Q})$ when $\delta(\nu_{\gamma,Q}) \in S$.

The *start configuration* of an MRCA $A = (S, \delta, s)$ is the configuration $\gamma_0 = (\{[0, 1]^2\}, \sigma)$ where $\sigma([0, 1]^2) = s$.

The *computation* of an MRCA $A = (S, \delta, s)$ is the orbit of γ_0 under δ^*, i.e., the infinite sequence $\gamma_0, \gamma_1, \gamma_2, \ldots$ in $CONFIG_A$, where each $\gamma_{t+1} = \delta^*(\gamma_t)$.

We have now specified the basic features of the MRCA model. Additional features may be added for specific purposes. For example, in Section 3.3 we use this model to color portions of $[0, 1]^2$ in a monotone way (i.e. without color changes once a cell's color is set). For this purpose, we define a *(monotone) k-coloring* MRCA to be a 4-tuple $A = (S, \delta, s, c)$, where

(i) (S, δ, c) is an MRCA;

(ii) $c : S \dashrightarrow [k]$ is the *state-coloring partial function*; and

(iii) for all $q \in dom(c), \nu \in N_S$, and $q' \in S$, if q' is $\delta(q, \nu)$ or a component of $\delta(q, \nu)$, then $q' \in dom(c)$ and $c(q') = c(q)$.

Condition (iii) ensures that δ never updates an already-colored cell in such a way as to alter or remove the color of any part of that cell. For each $i \in [k]$, the *set colored* i by a k-coloring MRCA $A = (S, \delta, s, c)$ is the set $S_i(a)$ defined in the now-obvious way.

Example 3.2.3. For each $n \in \mathbb{Z}^+$, let

$$\begin{aligned} Q_n &= Q\left(2n, \frac{4^n - 4}{3}, 0\right) \\ &= \left[\frac{1}{3}(1 - 4^{1-n}), \frac{1}{3}(1 - 4^{-n})\right] \times \left[0, 4^{-n}\right], \end{aligned}$$

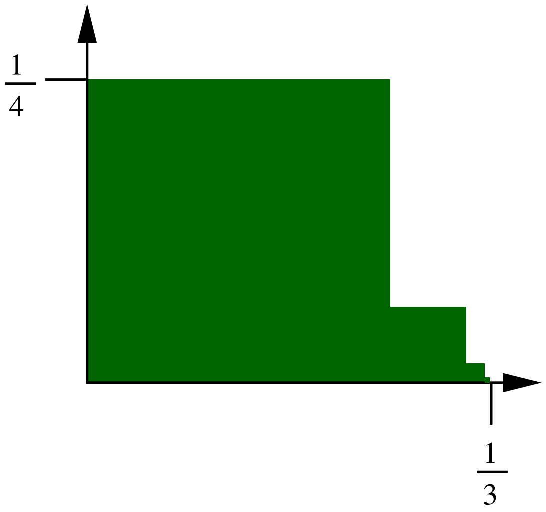


Figure 3.4 $\displaystyle\bigcup_{n=1}^{\infty} Q\left(2n, \frac{4^n-4}{3}, 0\right)$ colored in.

and let $X = \displaystyle\bigcup_{n=1}^{\infty} Q_n$. Consider the 1-coloring MRCA $A = (S, \delta, s, c)$, where $S = \{s, t, u, v\}$; $c(v) = 0$; $c(s), c(t)$, and $c(u)$ are undefined; and, for all $\nu \in N_S$,

$$\begin{aligned}
\delta(s, \nu) &= (t, t, u, t), \\
\delta(t, \nu) &= t, \\
\delta(u, \nu) &= (t, t, v, s), \\
\delta(v, \nu) &= v.
\end{aligned}$$

It is easy to see that $S_0(A) = X$, as depicted in Figure 3.4 with color 0 appearing as green.

3.3 MRCA Characterization of Computability

This section presents an MRCA characterization of computability under the small-boundary hypothesis of Section 3.1.

We define a set $X \subseteq [0,1]^2$ to be *MRCA-computable* if there exist open sets $G \subseteq X$ and $H \subseteq [0,1]^2 \smallsetminus X$, with $G \cup H$ dense on $[0,1]^2$, and a 2-coloring MRCA A such that $S_1(A) = G$

and $S_0(A) = H$. With the convention of 0 being "red" and 1 being "green," this is the definition that we gave in the introduction.

Our proofs use the following coloring notion. We define a *(cellular) k-coloring* of $[0,1]^2$ to be a partial function $\psi : \mathcal{Q} \dashrightarrow [k]$ satisfying the consistency condition that, for all Q_1, $Q_2 \in dom(\psi)$,

$$Q_1 \cap Q_2 \neq \emptyset \Rightarrow \psi(Q_1) = \psi(Q_2).$$

We call $\psi(Q)$ the *color* of cell Q in the coloring ψ. For each $i \in [k]$, the *set colored i by ψ* is then

$$S_i(\psi) = \bigcup \{Q \mid \psi(Q) = i\}.$$

The following technical lemma is a central part of our argument for the main theorem of this chapter.

Lemma 3.3.1. *If ψ is a cellular k-coloring that is computable, then there is a k-coloring MRCA A such that, for each $i \in [k]$, $S_i(A) = S_i(\psi)$.*

We now prove the main result of this chapter, deferring proof of this lemma.

Theorem 3.3.2. *If $X \subseteq [0,1]^2$ is a set whose boundary is computably nowhere dense, then X is computable if and only if X is MRCA-computable.*

Proof. Let $X \subseteq [0,1]^2$ be a set whose boundary is computably nowhere dense.

Assume that X is computable. Then, by Theorem 3.1.2, there exist computably open sets $G, H \subseteq [0,1]^2$ such that $G \cup H$ is dense on $[0,1]^2$ and X is a separator of G and H. Since G and H are computably open, there exist c.e. sets $\mathcal{A}_G, \mathcal{A}_H \subseteq \mathcal{B}$ such that $G = \bigcup \mathcal{A}_G$ and $H = \bigcup \mathcal{A}_H$. By standard techniques, each open ball $B \in \mathcal{B}$ can be written as a countable union of dyadic squares $Q \in \mathcal{Q}$. Since this process is effective, it follows that there exist c.e. sets $\mathcal{R}_G \subseteq \mathcal{Q}$ and $\mathcal{R}_H \subseteq \mathcal{Q}$ such that $G = \bigcup \mathcal{R}_G$ and $H = \bigcup \mathcal{R}_H$. Since G and H are disjoint open sets, every cell in $\mathcal{R}_G$ must be disjoint from every cell in $\mathcal{R}_H$. Hence the partial function $\psi : \mathcal{Q} \dashrightarrow [2]$ defined by

$$\psi(Q) = \begin{cases} 1 & \text{if } Q \in \mathcal{R}_G \\ 0 & \text{if } Q \in \mathcal{R}_H \\ \text{undefined} & \text{otherwise} \end{cases}$$

is a well-defined cellular 2-coloring of $[0,1]^2$. Clearly, $S_1(\psi) = \bigcup \mathcal{R}_G = G$ and $S_0(\psi) = \bigcup \mathcal{R}_H = H$. Moreover, since $\mathcal{R}_G$ and $\mathcal{R}_H$ are c.e., ψ is computable. It follows by Lemma 3.3.1 that there is a 2-coloring MRCA A such that $S_1(A) = S_1(\psi) = G$ and $S_0(A) = S_0(\psi) = H$.

Conversely, assume that X is MRCA computable. Then there exists computably open sets $G \subseteq X$ and $H \subseteq [0,1]^2 \setminus X$, with $G \cup H$ dense on $[0,1]^2$. Then X is a separator of G and H, so X is computable by Theorem 3.1.2. □

Proof of Lemma 3.3.1.

We now discuss how to compute a k-coloring MRCA A such that $S_i(A) = S_i(\psi)$ for any $i \in [k]$ and computable cellular k-coloring $\psi : \mathcal{Q} \dashrightarrow [k]$. Since ψ is computable, there exists a Turing machine M with right-infinite tape that identifies the correct color of that square with respect to ψ (if it exists) given an encoding of a dyadic-width square $Q \in \mathcal{Q}$ by changing to that color at the conclusion of its computation[†].

We translate M into a one-dimensional CA that extends to the right. It is clear that a one-tape Turing machine with infinite cells to the right can be simulated by a CA with infinite cells to the right. We call the state of the cells to the right of computation *cookie states* c for reasons to be explained later. We also slightly alter that construction as follows.

(1) Add an extra cell to the left end of the CA. This reserves space to assist in coloring and provides a buffer between units of computation. Call this the “rabbit hole” [8] as it becomes smaller and smaller as computation progresses (or as we “fall into” it).

[†] Note that “coloring” in this view is an infinite process as the machine has an infinite tape.

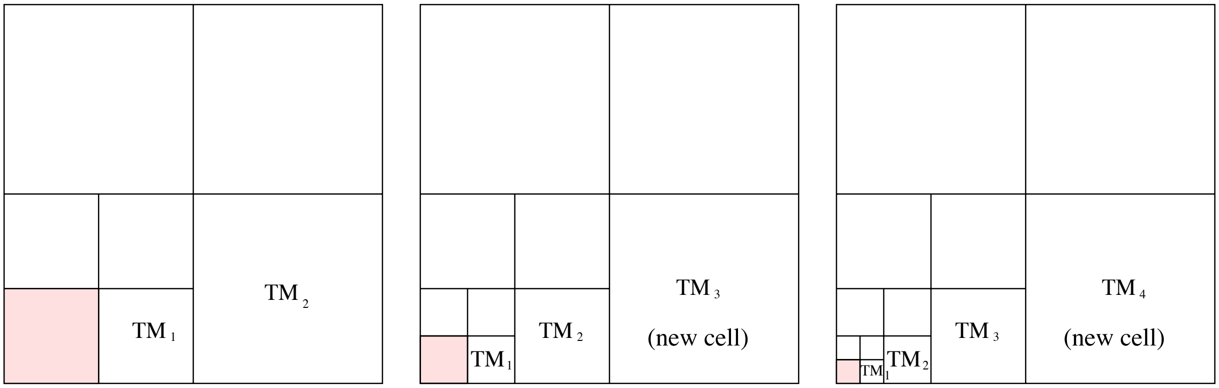


Figure 3.5 Use and creation of space for Turing machine computation by "falling down a rabbit hole." Note that the rabbit hole is the only cell to fission each time while the rightmost cell is where new computation can appear.

(2) Lay out the remaining cells such that each cell is twice the length and width of the cell to its left and half the length and width of the cell to its right.

(3) Add rules to allow for both periodic and computation-prompted fission. Importantly, the latter rules should not extend to fissioning to create more cells to color.

(4) Add rules to pass a copy of the address information off the right end of computation.

We call such a modified rule set a *computational unit function* C_M and denote its operation on an input dyadic square $Q(n, i, j)$ by $C_M(Q(n, i, j))$. Reserve the "rabbit hole" cell as described in **(1)** above and lay out the remaining cells according to **(2)**. This setup is shown in Figure 3.5. Reserve the last cell on the right as a signal cell for fission computation. This initial configuration given encoded input for any dyadic square $Q(n, i, j)$.

We now address the two, more complicated modifications **(3)** and **(4)** in more detail. The former creates more space for the current and child computational units (via prompted and periodic fission, respectively) while the latter guarantees that computation $C_M(Q(n, i, j))$ will eventually occur for any $n, i, j \in \mathbb{N}$.

Fission for more computational space is prompted when a computational unit attempts to use the last space on the right for non-coloring reasons. This triggers a signal being sent left to the “rabbit hole” which fissions. This is shown in Figure 3.5 as the pink cell splitting and taking on the state of the cell to the right as it’s lower-left cell.

At the next step, the cell to the right of the rabbit hole then reads that the rabbit hole has fissioned and taken on its state. Because the signal to make more space had to come through this cell to arrive at the rabbit hole, computation has already frozen to complete this process. So the cell to the right of the former rabbit hole can take on the state of the cell to its right. In this way, every cell eventually takes on the state of the cell to its right except the last cell, which now can take on the state dictated by C_M.

Periodic fission can be handled similarly by selecting a transition that occurs with guaranteed regularity in the computation of C_M and instead send a signal state exactly as if fissioning for more computational space before continuing computation. If there is no such naturally-occurring transition, we can easily add a counter to the end of C_M to trigger this fission. The signal is sent to the left end of the computational unit and triggers the same set of transitions as when the computational unit requires more space.

The only difference from fission for computational reason is that, when the fission process is complete, we change the now-empty rightmost cell to the cookie state c. This cell will now not be used for computation and, if additional space is required by the computation, it will fission if there is only cookie states to its right. However, if the computation concludes that $C_M(Q(n,i,j))$ should be colored k, these cookies states will also change that color — they serve the function of marking the right edge of this computational unit.

The process of passing address information as per modification **(4)** from the view of the computational unit is straightforward. Simply add a set of rules to send a copy of the address information to the right before the computation described by M begins. The process of initializing a computational unit with input will be written so that there is enough space to do this without additional fission. This address information is then used by A, the full MRCA rule set, to initialize new computational units.

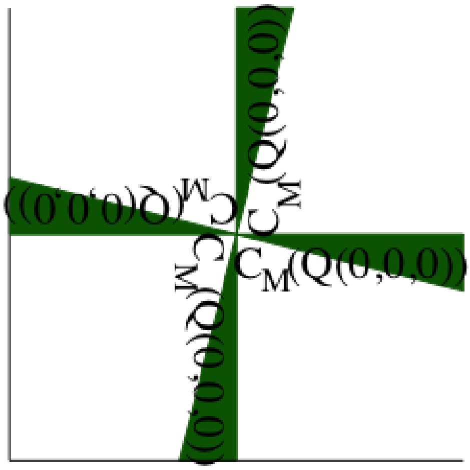


Figure 3.6 Initial state of each C_M with initial state $C_M(Q(0,0,0))$ rotated to show the direction of computation in each computational unit. The colored area is the space of that unit's computation.

Before describing the process of new computational unit creation, we pause to describe how the one-dimensional computational units $C_M(Q(n,i,j))$ are laid out on the two-dimensional grid. Four identical computational units $C_M(Q(n,i,j))$ are laid out as shown by the orientation of $C_M(Q(0,0,0)$ in Figure 3.6 for each input $Q(n,i,j)$. The right computational unit is computing left to right and each of the other three units are rotated 90 degrees clockwise.

All rules included in C_M are also rotated 0, 90, 180, and 270 degrees and new symbols are used for each direction (e.g. 0 becomes 0_0, 0_{90}, 0_{180}, and 0_{270}) to get a consistent set of rules for each direction. For simplicity, we will continue to discuss computational units as if they are oriented left-to-right (e.g., Figure 3.5).

The MRCA A is initialized with four computational units as shown in Figure 3.6 and the first set of rules for A concerns taking the address information available at the right end of each computational unit and using that information to initialize the child computational units. This state is shown in more detail for one computational unit $C_M(Q(1,0,0))$ in Figure 3.7.

A starts the process with transitions of cells above the cell marked u to fission to an

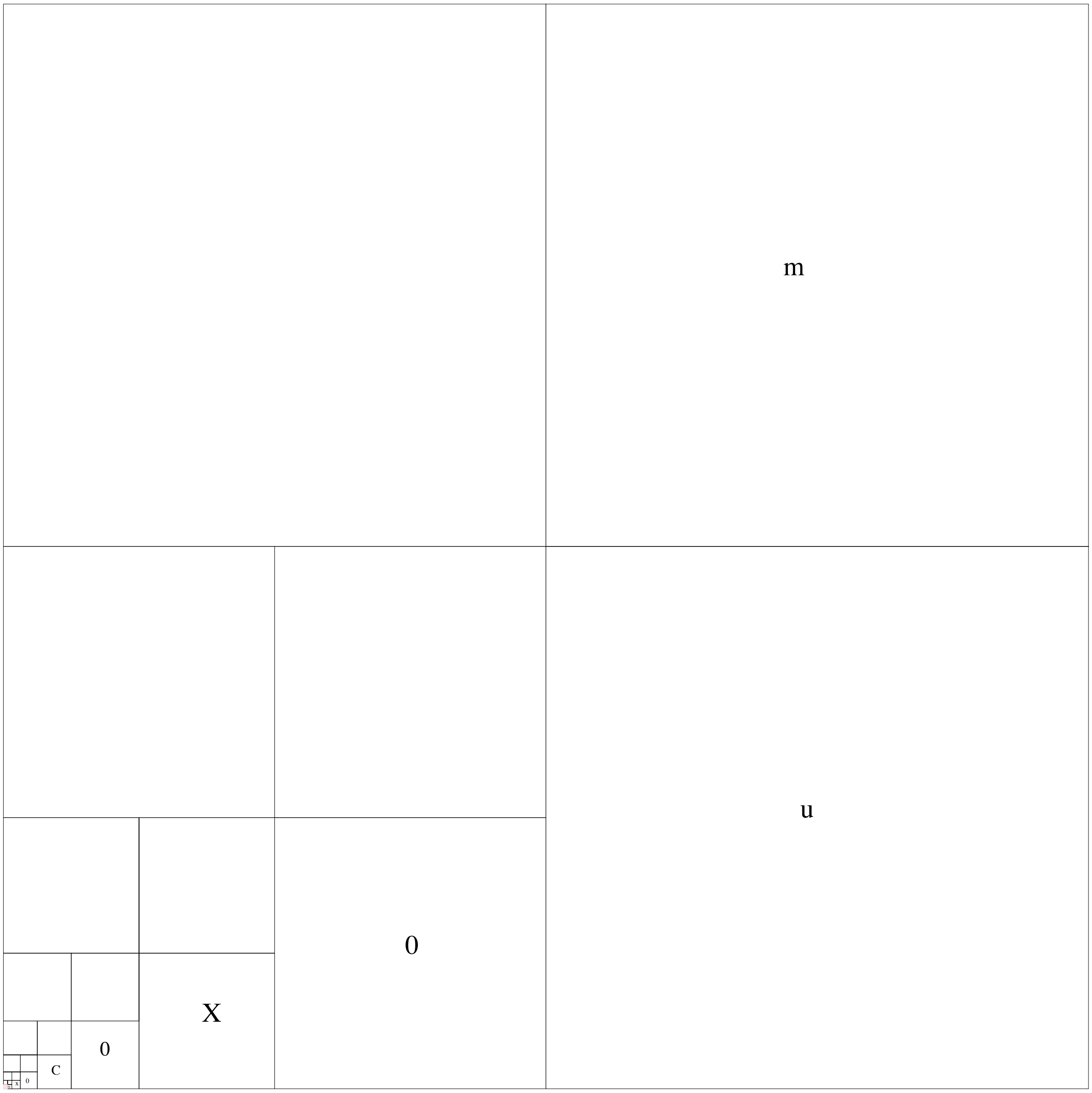


Figure 3.7 Complete initial state of $C_M(Q(1,0,0))$ including address (with x and y coordinates separated by an X), state C identifying the right edge of computation, a copy of the address for passing, a signal cell u to start passing address information up, and a cell in state m to denote the vertical center of any future child pinwheel (which would be 4 computational units for $C_M(Q(2,1,1))$ in this orientation). Note that the space above the current computational unit contains a mark where the center of the dyadic cell above $C_M(Q(1,0,0))$ appears. All cells to the right of the C cell will become cookie cells. Note that, unfortunately, the left end quickly becomes unreadable.

appropriate size to read the incoming address state information. This information is sent upwards until meeting the mark placed during initialization of this computational unit is found.

The address information is then used by rules in A to construct a new set of four computational units with slightly different addresses as well as set marks . Specifically, the computational unit for $Q(n, i, j)$ generates computational units to compute $Q(i+1, 2i+1, 2j+1)$, $Q(i+1, 2i, 2j+1)$, $Q(i+1, 2i, 2j)$, and $Q(i+1, 2i+1, 2j)$. These four computational units appear as shown in Figure 3.8, where each "parent" unit sends address information north (relative to its rotation). Note that, by this construction, a north-to-south child computational unit extends back to the parent computational unit and shares a cell in a cookie state as part of their computational space. This non-interference is part of why we reserve cells in cookie states.

The reason we create duplicate, rotated computational units and set cells to a cookie state is to simplify the process of coloring. There are two processes used to color cells after a computational unit has decided a color for space $Q(n, i, j)$. A cell will be colored when either of the following occurs.

(A) A cell is part of the space of a computation that concluded $Q(n, i, j)$ should be colored.

(B) A cell is the neighbor of two colored squares that differ from it horizontally and vertically with the exception of rabbit hole cells.

When a computational unit concludes that input square $Q(n, i, j)$ must be colored, the signal is sent along the computational unit's cells away from the rabbit hole i.e., to the right in left-to-right computation. This signal and coloring continues to the edge of $C_M(Q(n, i, j))$ through the cookie states, completing coloring process **(A)**. When four rotated computational units finish this coloring process, we are left with colored lines of cells as shown in red in Figure 3.9. Note that there are "shared" cookie cells between parent and one child so

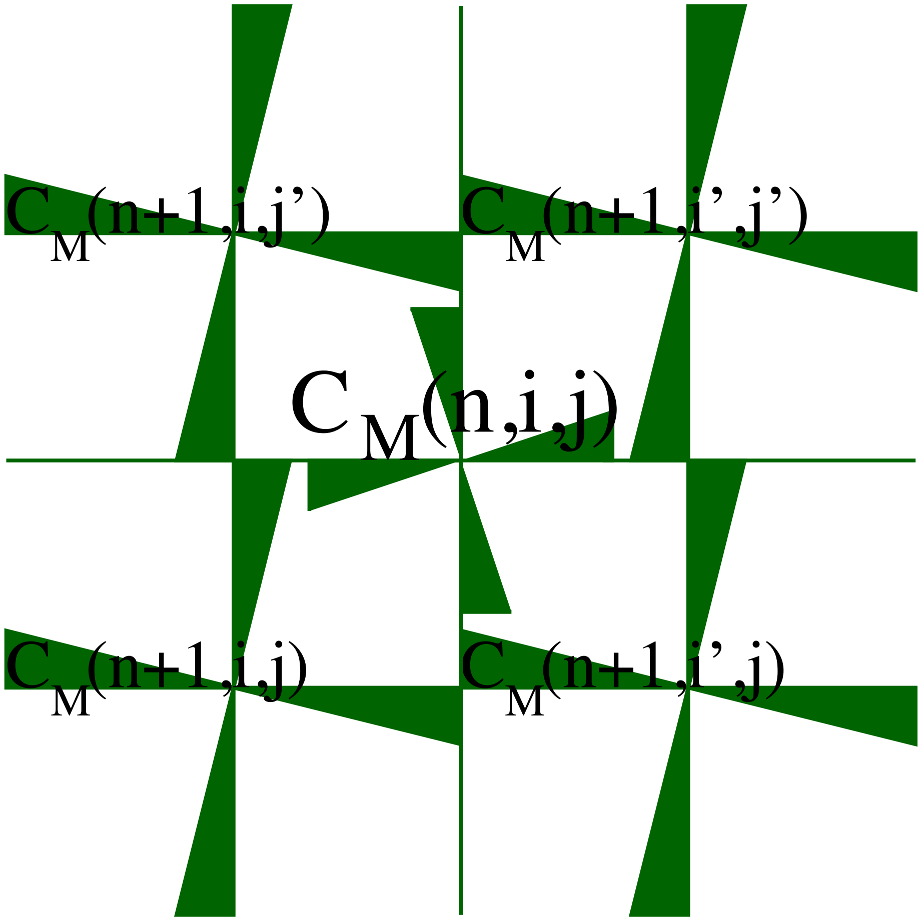


Figure 3.8 How the child computational units of $C_M(Q(n,i,j))$ are laid out where $i' = 2i+1$ and $j' = 2j+1$. The green lines along each axis emanating from the parent computational unit represent cookies states. Although these lines appear to cross the child units, the child units will preserve the cookies of their ancestors.

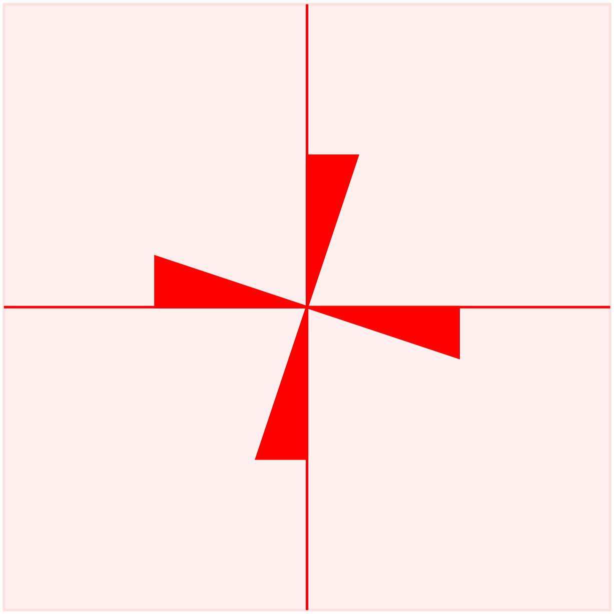


Figure 3.9 Example coloring of a dyadic square with red indicating the computational and cookie states of the coloring computation and pink indicating the completion of coloring via neighbor examination.

the coloring of each computational unit must take into account the direction from which it received color information when changing color.

To complete coloring the dyadic square, our MRCA A has a general rule **(B)** that, if a cell has an k-colored neighbor that differs from it horizontally and another k-colored neighbor that differs from it vertically, it becomes that color. If one computational unit concludes its input dyadic square will be colored, all of them will because they are performing the same computation. Starting with the rabbit hole of each unit, each of the cells will have a north-south neighbor and an east-west neighbor that change color as part of this computation. Initially, the cells adjacent to two "rabbit holes" will be colored followed by its neighbors in two directions and so on until only and exactly the required dyadic square is colored. This is shown in Figure 3.9 in pink.

An important exception to the coloring by process **(B)** is that rabbit holes never change color according to process **(B)**. A rabbit hole will not change color for any other combination of neighbor colorings to ensure the rabbit hole is colored only by its own computational unit or that of a child computational unit covering this computational unit's space (i.e., not a neighboring computational unit or any other combination of child, sibling, or nephew computational units).

We now briefly discuss the case of a child computational unit coloring its square when the parent computation has not concluded. For example, in the lower-left corner of Figure 3.8, $C_M(Q(n+1,i,j))$ may conclude before $C_M(Q(n,i,j))$ does. To see our approach colors only the correct child dyadic-width square and no others, we consider two cases.

(1) Parent $C_M(Q(i,x,y))$ will continue computing forever.

(2) Parent $C_M(Q(i,x,y))$ will result in a coloring.

In case **(1)**, there is no loss in coloring the computational area of the child computation because it will not result in coloring the entire square $Q(n,i,j)$: other child computational units of $C_M(Q(n,i,j))$ will never result in a coloring with process **(A)** by assumption nor with process **(B)** because the rabbit hole will never change color. Even in the case of $C_M(Q(n+1,2i,2j))$ and $C_M(Q(n+1,2i+1,2j+1))$ in Figure 3.8 changing color, the rabbit holes of $C_M(Q(n+1,2i+1,2j))$ and $C_M(Q(n+1,2i,2j+1))$ will be uncolored to prevent the coloring of the entire square. Thus, neither methods of coloring will color other, incorrect child computational units.

In case **(2)**, the child computational unit must be colored the same color as $C_M(Q(n,i,j))$ by ψ. Coloring of the dyadic square of that child computational unit will halt the computational unit of $C_M(Q(n,i,j))$ in that space. When the other three computational units for $C_M(Q(n,i,j))$ begin coloring process **(A)**, they can use the coloring resulting from the child's coloring to color the rest of the space $Q(n,i,j)$ using coloring process **(B)**.

It is reasonable to ask whether the fissions caused by child computation can "outpace" the speed at which cells are colored. We can guarantee this coloring will eventually occur because the fissions prompted by computational units cannot exceed the speed of the coloring process. Thus each child computational unit will be able to color its computed dyadic square.

Through this procedure, the entire unit square will be colored green or red by computation of dyadic squares in the limit of these MRCA rules. $S_i(A) = S_i(\psi)$ for any $i \in [k]$ and computable cellular k-coloring $\psi : \mathcal{Q} \dashrightarrow [k]$ because the computational units of A compute $S_i(\psi)$ and color regions i in dyadic squares as needed. □

3.4 MRCA Characterization of Polynomial-time Computability

Given our main result in Theorem 3.3.2, it is natural to ask: How does the set of poly-time computable sets [5] relate to the set of MRCA poly-time computable sets? In order to discuss feasible MRCA computation, we use the idea of poly-time open sets.

Definition 3.4.1. *Given a set $X \subseteq [0,1]^2$, let d_H denote the Hausdorff distance between two sets and, for any $t \in \mathbb{N}$, let $\mathcal{A}_t \subseteq \mathcal{B}$ be a set of balls such that $\bigcup \mathcal{A}_t \subseteq X$ and $d_H\left(\bigcup_{B \in \mathcal{A}_t} B, X\right) \leq 2^{-t}$ (call $\mathcal{A}_t$ a* t-approximate set *of balls for X).*

A set $X \subseteq [0,1]^2$ is poly-time open *if there is a polynomial $p(n)$ such that for all $t \in \mathbb{N}$ we can decide if a ball $B(q, 2^{-t})$ is in $\mathcal{A}_t$ in time $p(t)$. That is, the set $\{B \mid B \in \mathcal{A}_t\} \in P$.*

Note that we are no longer enumerating balls (as in the definition of computably open) but rather deciding if the ball belongs in the set. This allows us to use the parallelism of MRCAs to its fullest in defining *MRCA poly-time computable* sets.

Definition 3.4.2. *$X \subseteq [0,1]^2$ is* MRCA poly-time computable *if there exists poly-time open sets $G \subseteq X$ and $H \subseteq [0,1]^2 \setminus X$, with $G \cup H$ dense in $[0,1]^2$, and a 2-coloring MRCA A such that $S_1(A) = G$, $S_0(A) = H$. All other points, including points on the topological boundary of X, remain uncolored throughout the execution of the MRCA.*

We now extend the definition of computably nowhere dense sets to poly-time nowhere dense sets.

Definition 3.4.3. *Z is* poly-time nowhere dense *if there is a poly-time function $f : \mathcal{B} \smallsetminus \{\emptyset\} \to \mathcal{B} \smallsetminus \{\emptyset\}$ such that, for all $B \in \mathcal{B} \smallsetminus \{\emptyset\}, f(B) \subseteq B \smallsetminus Z$.*

We are now equipped to discuss the feasible version of Theorem 3.1.2.

Theorem 3.4.4. *If $X \subseteq [0,1]^2$ is a set whose boundary is poly-time nowhere dense, then the following two conditions are equivalent.*

(1) X is computable in poly-time.

(2) X is a separator of two poly-time open sets whose union is dense.

Proof. First note that, by the poly-time version of Observation 3.1.1 [31][‡], there is a poly-time open dense set $D \subseteq [0,1]^2$ such that $D \cap \partial X = \emptyset$. Since D is poly-time open, there is a set $\mathcal{D} \subseteq \mathcal{B}$ such that $D = \bigcup \mathcal{D}$. Since $D \cap \partial X = \emptyset$ and $\{X^\circ, ([0,1]^2 \smallsetminus X)^\circ, \partial X\}$ is a partition of $[0,1]^2$, it must be the case that every ball $B \in \mathcal{D}$ satisfies $B \subseteq X^\circ$ or $B \subseteq ([0,1]^2 \smallsetminus X)^\circ$.

(2) ⇒ (1): Assume the hypothesis of poly-time open sets $G, H \subseteq [0,1]^2$ such that X is a separator of G and H and $G \cup H$ is dense on $[0,1]^2$. Let $\mathcal{A}_{G_r}$, $\mathcal{A}_{H_r} \subseteq \mathcal{B}$ be the set of balls witnessing that G and H are poly-time open given a radius r. Let $f : (Q \cap [0,1])^2 \times \mathbb{N} \to \{0,1\}$ be computed by Algorithm 4.

Note that this is nearly the same function f appearing in Algorithm 1 except that we iterate $B \in \mathcal{D}$ rather than $B \in \mathcal{B}$ which has no effect on the correctness of f. We omit further argument of the correctness of the function based on the correctness of Algorithm 1. Each check of $B \in \mathcal{A}_{G_r}$ and $B \in \mathcal{A}_{H_r}$ takes polynomial time in r and D itself is poly-time open so f can be computed in poly-time.

(1) ⇒ (2): We now show that the set

$$\mathcal{D}_G = \{B \in \mathcal{D} \mid B \subseteq X^\circ\}$$

[‡] In [31], a more general notion of Γ-nowhere dense over languages is used. If we set Γ to the set of polynomial time functions and generalize the results from languages to the set $[0,1]^2$, a poly-time version of Observation 3.1.1 follows.

```
Algorithm 4   $f(q,r) = y$
              Input: $(q,r) \in (Q \cap [0,1])^2 \times \mathbb{N}$.
              Output: $y \in \{0,1\}$.
Find $B \in \mathcal{D}$ such that
(i) $B \in \mathcal{A}_{G_r}$ and $B \cap B(q, 2^{1-r}) = \emptyset$
or
(ii) $B \in \mathcal{A}_{H_r}$ and $B \cap B(q, 2^{-r}) \neq \emptyset$
if (i) holds then
    return 1
else
    return 0
end if
```

is enumerable in poly-time given X is poly-time computable in order to show that $G = \cup\mathcal{D}_G$ is poly-time open. We know from the proof of Theorem 3.1.2 that

$$\mathcal{D}_G = \{B(q,r) \in \mathcal{D} \mid r = 0 \text{ or} f(q, n(r)+1) = 1\}.$$

f is poly-time computable and we can find if $B \in \mathcal{D}$ in poly time because D is poly-time open.

$G = \cup\mathcal{D}_G$ is poly-time open because we can decide whether balls are in $\mathcal{D}_G$ in poly-time. Likewise, we can identify $\mathcal{D}_H$ as evidence that $H = \cup\mathcal{D}_H$ is poly-time open. The remaining correctness arguments at the end of the proof of Theorem 3.1.2 shows that (2) holds. □

To show our main result, we will need a similar technical lemma to Lemma 3.3.1 but this time providing a poly-time bound.

Lemma 3.4.5. *If ψ is a cellular k-coloring that computes $\psi(Q(n,i,j))$ in time $t(n)$ for any $i,j,n \in \mathbb{N}$, then there exists a $O(t(n)^2 + n^2)$-time k-coloring MRCA A such that (for all $m \in [k]$) $S_m(A(Q(n,i,j))) = S_m(\psi(Q(n,i,j)))$.*

We now state and prove the main result of this chapter, deferring proof of this lemma.

Theorem 3.4.6. *If $X \subseteq [0,1]^2$ is a set whose boundary is poly-time computably nowhere dense, then X is poly-time computable if and only if X is poly-time MRCA-computable.*

Proof. First, assume that X is poly-time computable. By Theorem 3.4.4, there exist poly-time open sets G and H whose union is dense and that X separates. Set $t \in \mathbb{N}$.

Because G and H are poly-time open, we can identify t-approximate sets of balls $\mathcal{G}_t \subseteq G$ and $\mathcal{H}_t \subseteq H$ as witness sets that G and H are poly-time open. As discussed in Theorem 3.1.2, any set covered by a set of balls can be covered by a countable union of dyadic squares. Call the set of such squares $\mathcal{R}_{G_t}$ and $\mathcal{R}_{H_t}$ for $\mathcal{G}_t$ and $\mathcal{H}_t$ respectively. By the same process used to query $\mathcal{G}_t$ and $\mathcal{H}_t$, it takes poly-time to query whether a given dyadic square is in $\mathcal{R}_{G_t}$ and $\mathcal{R}_{H_t}$ (respectively). Now note that $\mathcal{R}_{G_t}$ and $\mathcal{R}_{H_t}$ are disjoint so

$$\psi(Q) = \begin{cases} 1 & \text{if } Q \in \mathcal{R}_{G_t} \\ 0 & \text{if } Q \in \mathcal{R}_{H_t} \\ \text{undefined} & \text{otherwise} \end{cases}$$

is a well-defined cellular 2-coloring of $[0,1]^2$. $\mathcal{R}_{G_t}$ and $\mathcal{R}_{H_t}$ are decidable in poly-time because G and H are poly-time open so ψ is poly-time computable. It follows by Lemma 3.4.5 that there is a 2-coloring MRCA A such that, at time $p(t)$ for some polynomial p, $S_1(A) = S_1(\psi) = G_t$ and $S_0(A) = S_0(\psi) = H_t$. Our result follows.

Conversely, assume X is MRCA poly-time computable. Then X is a separator of poly-time open sets G and H. By Theorem 3.4.4, X is therefore poly-time computable. □

Proof of Lemma 3.4.5.

It suffices to show that, given a Turing machine M in the class required by the proof of Lemma 3.3.1 that computes $\psi(Q(n,i,j))$ in time $t(n)$, the MRCA A constructed from M given in the proof of Lemma 3.3.1 takes time $O(t(n)^2 + n^2)$ to color $Q(n,i,j)$.

We specifically compare the run time of M on an input $Q(n,i,j)$ to the amount of time it takes for MRCA A to color $Q(n,i,j)$ green or red given in the proof of Lemma 3.3.1. Note that, if M runs forever on input $Q(n,i,j)$, A will also run C_M on $Q(n,i,j)$ forever.

In order to compare the run time of M on input $Q(n,i,j)$ to the time it takes for A to color $Q(n,i,j)$, without loss of generality we make the following simplifying assumptions:

(A) Address information is sent from each computational unit immediately before computation begins. Although our earlier proof is correct as long as address information is sent at some point, this assumption simplifies time analysis without loss of generality.

(B) Each computational unit C_M's set of starting cells is simply the input ixj to compute $Q(n,i,j)$. Additionally, i and j are encoded in n bit binary strings ($Q(0,0,0)$ is encoded by an empty string for i and j). Although each computational unit C_M will typically require more space to perform computation, we can add a constant number of rules to C_M to create the initial space M uses and any additional computational space is included in either M's time $t(n)$ or the amount of time fission takes.

(C) The starting set of cells for each computational unit will take up at most one cell less than a quarter of the width of the dyadic cell it is computing. This means that all computation takes place in the lower-left quadrant of its child dyadic square, leaving space for a "pinwheel" of identical child computational units. This bounds the amount of actual space that each computational unit uses.

(D) Periodic fission of computational units will occur every $k \in \mathbb{N}$ steps. This is considered separately from fission for more computational space and does not change the operation of the construction of Lemma 3.3.1 (where fission is specified as simply periodic).

Because of how M's computation is transformed into a computational unit C_M, there are two classes of considerations: Slowdown due to M's translation to C_M and slowdown due to modifications **(1)** through **(4)** in the proof of Lemma 3.3.1. We will consider computational units in the standard orientation (e.g., as depicted in Figure 3.5). Units in other orientations are completed in parallel so no additional time is taken.

Fortunately, the direct translation of a right-infinite-cell Turing machine M to a one-dimensional CA results in no slowdown. Likewise, each rule for the CA can be translated into a rule for C_M that ignores cells to it's north and south.

Modifications **(1)** through **(4)** to create each C_M in the proof of Lemma 3.3.1 do cause a slowdown of C_M relative to Turing machine M. Modifications **(1)** and **(2)** simply reserve extra space and details a new layout to the cells so no slowdown in accrued. **(3)** directly slows the computation down by a factor of k to allow periodic fission so the amount of time that is spent computing the color of any $Q(n, i, j)$ is bounded by $kt(n)$.

As part of modification **(3)**, we also must consider how often computation will prompt fission. In the worst case, every one of the $t(n)$ computational steps will require a fission for more space. As per the description in the proof of Lemma 3.3.1, the signal must be sent from the right edge to the left, the rabbit hole fissions, and then all the cells shift one cell left before continuing. As the length of the entire unit is bounded above by $t(n)$, this takes at most $2t(n) + 1 = O(t(n))$ steps. If we perform this set of steps for each of the $t(n)$ steps of computation, the total time for fission for computation is $O(t(n)^2)$.

We also must consider the amount of time spent before the creation of the computational units that compute $Q(n, i, j)$. To decide inputs of precision n, appropriate ancestor computations of precisions 1 through $n-1$ must exist in A. For example, to create the computational unit for $Q(3, 1, 10_{binary})$, the ancestor computational units with inputs $Q(1, 0, 0)$ and $Q(2, 0, 1)$ must be created first. Note that, due to assumption **(A)**, these computations will not be complete before sending the address information onwards so we will only consider the time to pass address information to each child computational unit.

Let us consider an arbitrary ancestor computational unit computing $Q(m, k, l)$ and the process whereby a child computational unit computing $Q(m+1, k', l')$ is created where k' is $2k$ or $2k+1$ and l' is $2l$ or $2l+1$, depending on our final i and j respectively. The process

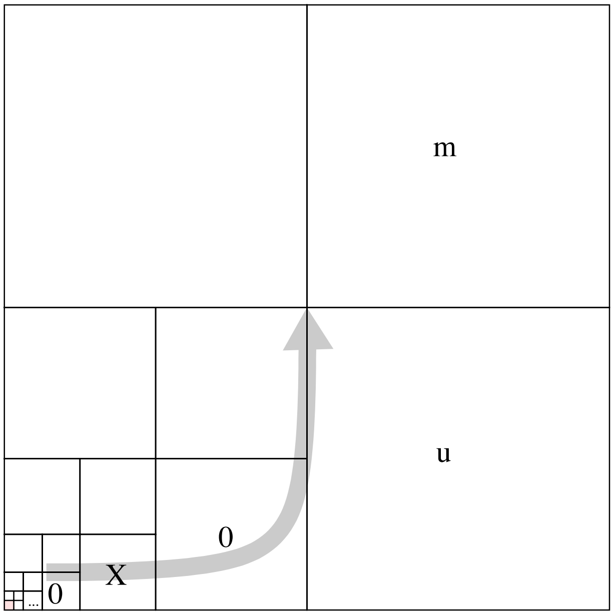


Figure 3.10 The initial configuration of the right computational unit $C_M(Q(1,0,0))$ before creation of the computational units for $Q(2,1,1)$. u denotes a signal state above and to the left of which the child "pinwheel" will appear. The gray arrow indicates the direction that the address information is passed and the mark m denotes the center of the child computational unit pinwheel placed by A during initialization of $C_M(Q(1,0,0))$.

of creating a child computational unit computing $Q(m+1,k',l')$[§] can be decomposed into three processes: passing the address to the right, passing the information up, and creating the "pinwheel" of child computational units. The (parent) computational unit has $2m+1$ cells before computation begins and does not include the cell directly below where the "pinwheel" of child computational units will be placed (due to the assumptions above). This initial setup is shown in Figure 3.10.

To address the first question of passing address information to the right, we need to consider what size the cells are in the parent computation as this will dictate how many steps it will take to send the left-most bit of the address to the point at which that information will be sent up. It takes at least $2m+1$ steps to pass this information over to the cell marked

[§] Note that this procedure focuses on the creation of the upper-right child "pinwheel" of computational units but the same arguments apply if we are interested in any other child pinwheel of computational units.

C in Figure 3.10 because the first bit of the original input must pass through X, Y, and the dividing bit (character X in assumption **(B)** above). As this happens before any other computation by assumption **(A)**, there are exactly $2m + 1$ states to pass.

The time to pass information up is $2(2m + 1)$ because each new address passed north to the west of the space marked u in Figure 3.10 will fission, causing a one step pause in the passing process for each passing step. Note also that, in order for initialization of the other computational units in the same child pinwheel, the other cells near the center of the pinwheel will also fission but this can be done in parallel with passing the address information.

Last, we analyze how long it takes to initialize a child pinwheel. Each child computational unit will have size $2m + 3$ as a bit is added to both x and y coordinates of the ancestor's address. As noted above, fission of the other "spokes" of the pinwheel of computational units must take place as address information is sent in order for communication to remain possible. Thus, the time to send the address information through the rabbit holes of the child computational units and initialize them is $2m+3$. Thus, the time to complete all three processes that end with a child pinwheel of computational units is $2m+1+2(2m+1)+2m+3 = 8m + 6$.

We now have what appears in the upper-right quadrant of Figure 3.8, the unrotated computational unit. The ancestor computational unit continues computation and the new, child computational units can begin by sending *their* address information to create a new child pinwheel of computational units.

Our original question was the length of time it takes to create and compute the color of an arbitrary $Q(n, i, j)$. To find this, we must take the summation of the time it takes each ancestor to send their address information to child computational units until the computational units for $Q(n, i, j)$ are initialized and complete. This summation considers starting with a cell with address information of size 3 (i.e., $0x0$, $0x1$, $1x0$, or $1x1$ depending on the identity of i and j) and finishes with ixj ready to compute. Since each increase in address size increases m by two (adding one bit each to the x and y coordinate), it is characterized by the summation

$$\sum_{m=3}^{2n+1} \begin{cases} 0 & \text{if } m \text{ even,} \\ 8m+6 & \text{otherwise,} \end{cases}$$

as only odd-length coordinates occur. Now note that this summation is equivalent to

$$3(2n+1)+2-(6+2)+\sum_{m=2}^{2n} \begin{cases} 0 & \text{if } m \text{ even,} \\ 8m+6 & \text{otherwise,} \end{cases}$$

which is equivalent to just doubling all values of m. Note that, by assumption **(A)**, $C_M(Q(n,i,j))$ has to send its address information to the east before starting its own computation so this creates a $2n+1$ step delay included below.

$$\begin{aligned} \left(\sum_{m=1}^{n} 16m+12\right)+18n-7 &= 16\sum_{m=1}^{n} m+12n+18n-7 \\ &= 16\left(\frac{n^2+n}{2}\right)+30n-7 \\ &= 8n^2+38n-7 \\ &= O(n^2). \end{aligned}$$

So the full run time of getting to the computation of $C_M(Q(n,i,j))$ and then complete that computation is $O(t(n)^2+n^2)$. □

3.5 Similar Work to the MRCA

In this section we consider three classes of related work: notions similar to nowhere dense sets used to characterize "small" boundaries, cellular automaton models similar to the

MRCA using fission, and cellular automaton models with a name similar to or exactly the same as the MRCA.

We note that Myrvold [36] and Parker [40] have considered notions somewhat similar to MRCA-computability. (See also [41].) Myrvold's ”decidability ignoring boundaries” differs in that there is no small-boundary hypothesis, and the sets G and H are required to be the interiors of X and its complement, respectively. The latter requirement is, for our purposes here, unduly restrictive: We show in the Turing-gapped comb of Section 3.1 a computable set with computably nowhere dense boundary whose interior is not computably open. Parker's ”decidability up to measure zero” requires the set of points not correctly ”decided” to be small in a measure-theoretic sense, while we require it to be small in a topological sense.

Other cellular automaton models use concepts similar to fission to address issues other than real computation. Zeno machines [11] (or Accelerating Turing Machines) are Turing machines that take 2^{-n} steps to complete the n^{th} computational step. Infinite time Turing machines [18] (ITTMs) generalize Zeno machines by allowing a *limit state* wherein every cell is set to the limit supremum of the values previously displayed in that cell. These models subdivide time in a way similar to how MRCAs subdivide space. A Scale-Invariant Cellular Automata (SCA) [50] create a lattice graph of cells where each level is half the width of the preceding level. Cells at depth i perform computation every 2^{-i} time units. In contrast, the MRCA sets cell size more non-uniformly than according to depth in a lattice and all cells update at each time step.

There are a number of similar models to the MRCA applied to other areas of research as surveyed by Dunn **??**. Specifically, Kiester and Sahr [21] implement a Discrete Global Grid Systems (DGGS) [49] that uses the concept of cells of different sizes communicating but each cell of a given size appears on its own layer and is able to read cells both below and above it (i.e., larger and smaller cell sets). They generalize this idea to computation over incomplete topologies (that is, some cells are designate “live” while others are “dead”) so that each cell may have a different number of neighbors. When applied to the area of urban systems [21] and landscape ecology **??** this allows us to focus on areas of interest with more, smaller cells while computing very little over less interesting areas. They also generalize to hexagonal cells and overlay their topologies on spheres.

This model is similar to a supervised version of the MRCA in that the topology of cells is set from the outset. Thus, their rules need only be concerned with state changes rather than also including fission instructions. We also simplify the model in a sense by changing the lower-resolution cell into its child, higher-resolution cells rather than letting both exist at the same time and be able to communicate with each other.

There are also early models by Nagel [37; 38] that allow us to consider cells of multiple resolutions at the same time to simulate increasing traffic density. However, these approaches seem to be ad hoc mechanisms using probabilistic cellular automata of varying density rather an a general framework for computation.

CHAPTER 4. MULTI-RESOLUTION CELLULAR AUTOMATA SIMULATION

4.1 Introduction to the MRCA Simulator

We now show how the MRCA model defined in Chapter 3 can be actualized in software. We use the Java programming language (version 1.6) to build a simulator that takes as input an initial state and a set of MRCA transitions (specified by what we call *rules*) and creates an MRCA based on that input. This simulator is designed so both a casual user can easily write MRCA rules for simple computation and a more ambitious user can implement the constructions discussed in Chapter 3 and more.

4.1.1 Simulator Interface

The simulator starts in step 0 with an interface similar to that shown in Figure 4.1. The exact identity of the state of the initial cell depends on the rule file we will discuss shortly. The input rules file must be named `MainConstruction.txt` by default but any file name can be used if provided as a command-line parameter to the program.

The simulator runs in two threads, one for computing MRCA configurations and the other for displaying them. The graphical interface is the primary way in which the user examines MRCA configurations. The computational thread interacts with the user directly only by sending errors to the command line if there is a problem with the rule set.

The main portion of the graphical interface is a view of the current MRCA configuration.

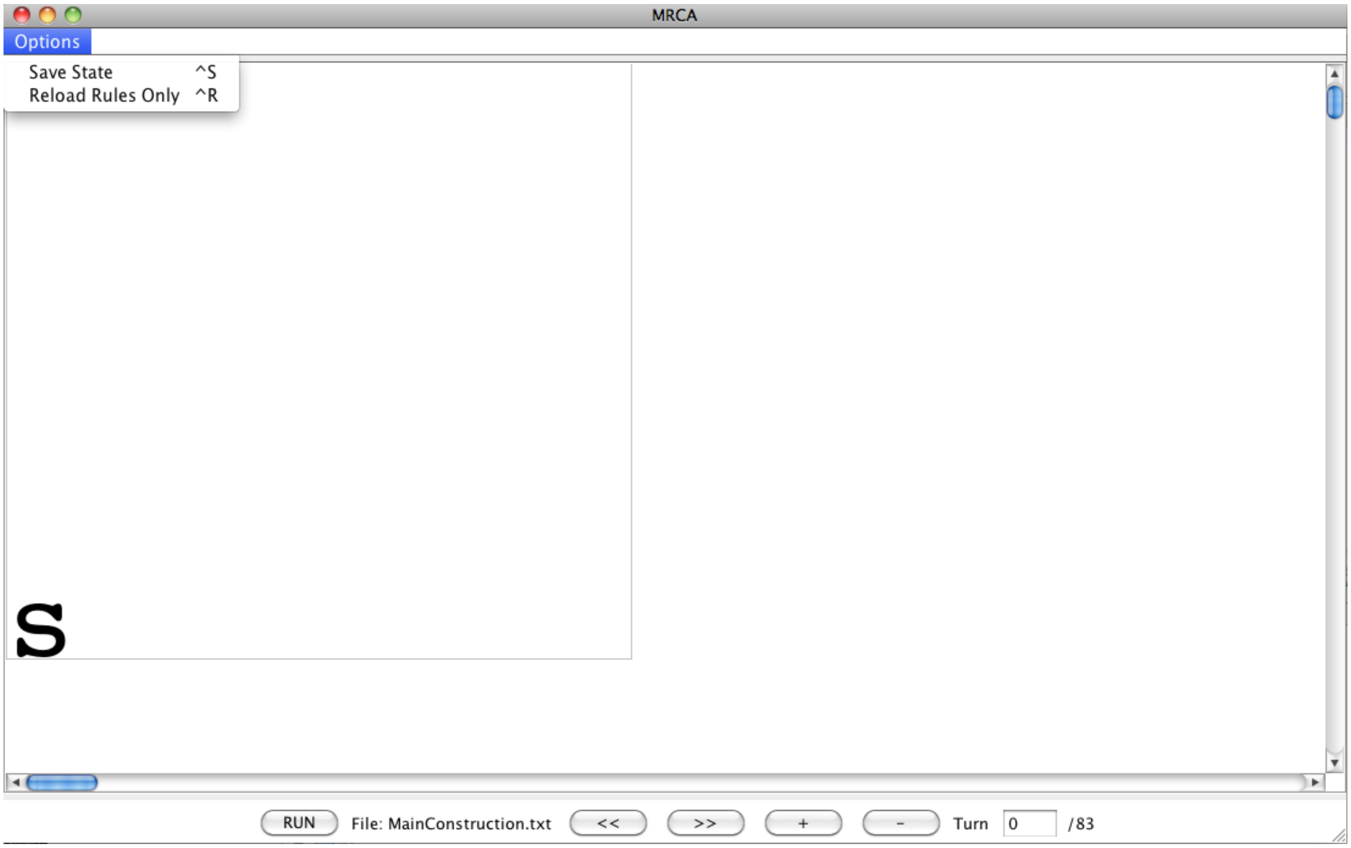


Figure 4.1 Example screenshot of the MRCA simulator at startup (scaled down) in Mac OS X 10.6.7.

Cells are edged in grey with the state names in black in the lower-left corner of each cell. In Figure 4.1, it is a single cell with state s. Along the extreme bottom from left to right is: a button to make the graphical interface "run" by showing successive configurations of the MRCA specified by the input rules; the file name of the input rules, buttons to view the previous and next configuration (`<<` and `>>`, respectively); buttons to increase and decrease the zoom factor with which we are viewing the cells (`+` and `-`, respectively); and a turn counter text box. The turn counter is automatically updated when you change configuration and can be used to jump between configurations. The turn number after the slash indicates which turn the computational thread has computed so far (in Figure 4.1, step 83).

In the upper-left corner is an Options menu with two items: Save State sends information about the current configuration to the command line that you launched this program from (also accessible via Control-S) and Reload Rules Only erasing the current rule set and reloads them from file (also accessible via Control-R). The last item is useful but tricky when developing a rule set as all configurations after the current one is erased but not any preceding configurations. This saves time by not recalculating all prior configurations but introduces the possibility that configurations are displayed that cannot be generated by the current rule set (i.e., were generated by a previously-loaded rule set).

One can also interact with the main view of the cells directly. Along the lower and right edge are scroll bars and we can also scroll using the mouse by clicking, holding, and dragging in the opposite of the desired direction. Note that the screen will not update in real time, instead waiting until you release the mouse, to avoid jitter on some computers. Last, you can zoom in by double-clicking on an area of the screen.

Those interested in using the simulator for more than casual development are advised to read the first 50 lines of the Java class `MRCAControlPanel` as shown in Appendix A. These options specify the following parameters of the program:

- `SKIP_TO_STEP` specifies the first step that the user will see upon loading the simulator.

The simulator will attempt to load this step immediately upon program start and send an error to the command line if it cannot. The computational thread of the simulator will continue to generate more configurations than this count.

- `LAST_STEP` specifies the last configuration that the computational thread will compute. We include this option because the computation thread will normally continue to compute configurations forever. If the rule set is sound, this will cause it to use more memory and CPU time over time to complete each configuration computation. Setting this option to a positive number causes the simulation to stop at the specified step instead.

- `WIDTH, HEIGHT` specifies the width and height of the control area at the bottom of the screen.

- `DRAW_WIDTH, DRAW_HEIGHT` specifies the width and height of the configuration display.

- `fontWidth` specifies the maximum font height for state symbols. If state symbols are too small (large), increase (decrease) this value (respectively) to obtain more viewable symbols.

- `viewOffsetX, viewOffsetY`, and `initialZoom` specify initial viewing area and zoom.

- `RUN_DELAY` specifies how long (in milliseconds) to wait between displaying configurations when the user clicks the RUN button.

4.1.2 Simulator Rule File Format

In order to specify the behavior of the MRCA, the user provides a rule file that is primarily composed of rules detailing groups of MRCA transitions. The simulator in the previous section will load and apply them to the initial and succeeding configurations. Specifically, the class InputReader deals with processing the rule files and Configuration2D applies them to the start and succeeding states (both classes in Appendix A).

Any line in the rule set file will be interpreted as a comment when it starts with a %.

There are two sections of the file before rules can be listed. First, the initial state of the MRCA is given in semi-colon delimited format with each line being a row of the MRCA and the configuration terminated by the word END. In our case,

```
s;
END
```

means that the MRCA starts with a single cell with state s. Only configurations with 2^n rows and columns (i.e., a uniform cell fission count and size) can be specified in the simulator given in Appendix A. The next section is used by the software to specify the color of cells in certain states (again terminated by an END). In our case,

```
(255,0,0)={RED}
(0,255,0)={GREEN}
END
```

means that cells in state RED and GREEN appear colored with RGB values $(255, 0, 0)$ and $(0, 255, 0)$, respectively.

We now show the context-dependent grammar for producing rules. Non-terminal symbols are presented in angle brackets <> and terminals presented without. We use / to separate options for productions, parentheses to group items, and a Kleene $^+$ to indicate that one or more of a set of productions with $^+$ next to it must be used as part of the current production. None of these items appear in the actual language.

$$\begin{aligned}
<\mathit{start}> &\rightarrow <\mathit{rule}> \,/\, <\mathit{orderedrule}>\\
<\mathit{rule}> &\rightarrow <\mathit{initialsyms}>; <\mathit{syms}>; <\mathit{syms}>; <\mathit{syms}>; <\mathit{syms}>; <\mathit{finalsyms}>\\
<\mathit{syms}> &\rightarrow \{(<\mathit{symbol}>;)^{+}\}/ <\mathit{symbol}> / * /| <\mathit{syms}>, <\mathit{syms}> |\\
<\mathit{initialsyms}> &\rightarrow <\mathit{symbol}>\\
<\mathit{finalsyms}> &\rightarrow <\mathit{symbol}> /| <\mathit{syms}>; <\mathit{syms}>; <\mathit{syms}>; <\mathit{syms}> |\\
<\mathit{symbol}> &\rightarrow <\mathit{literal}> \,/\, <\mathit{alias}>\\
<\mathit{orderedrule}> &\rightarrow <\mathit{oinitial}>; <\mathit{osyms}>; <\mathit{osyms}>; <\mathit{osyms}>; <\mathit{osyms}>; <\mathit{ofinal}>\\
<\mathit{osyms}> &\rightarrow [(<\mathit{symbol}>;)^{+}]/\{(<\mathit{symbol}>;)^{+}\}/ <\mathit{symbol}> / * /| <\mathit{syms}>, <\mathit{syms}> |\\
<\mathit{oinitial}> &\rightarrow [(<\mathit{symbol}>;)^{+}]\\
<\mathit{ofinal}> &\rightarrow [(<\mathit{symbol}>;)^{+}]
\end{aligned}$$

where productions for $<\mathit{literal}>$ and $<\mathit{alias}>$ depend on the rule set (the following section defines an example set of rules) and productions for $<\mathit{oinitial}>$ and $<\mathit{ofinal}>$ are dependant in that the same number of symbols must appear in each set for a given rule.

We now discuss the semantics of a rule set. A rule or ordered rule is read as a semicolon-delimited list of: initial state (set), north neighbor state (set), east neighbor state (set), west neighbor state (set), south neighbor state (set), and result state (set). A rule applies to a cell when the initial state and the cell's neighborhood state are members (in order) of the rule's first five entry sets. The cell's state then becomes the result state. As a simple example, the rule `x;a;b;c;d;y` specifies that a cell in state `x` with north neighbor `a`, east neighbor `b`, south neighbor `c`, and west neighbor `d` will transition to state `y`. A more complex example involving sets is that of `0;{ a,b};c;d;{ e,f};0` where a cell in state `0` north neighbor either `a` or `b`, east neighbor `c`, south neighbor `d`, and west neighbor `e` or `f` would transition to state `0`.

To specify the $b = 0$ and $b = 1$ neighbor states separately, we write $| <\mathit{syms}>, <\mathit{syms}> |$

where the first symbol matches the neighbor state at $b = 0$ and the second matches the neighbor state at $b = 1$ in the appropriate direction according to the definition provided in Section 3.2. For example, the rule `x;*;|a,b|;*;d;y` applies when the state is `x`, the east neighbor reads an `a` as the state of the lower neighbor ($b = 0$) and `b` as the state of the upper neighbor ($b = 1$), and `d` is the state of both neighbor indices to the west. When `*` appears as a neighbor state, it matches any neighborhood set. The rule will match any state in that direction. Note that it cannot appear as the initial state or result state.

<*finalsyms*>'s final production specifies how fissions are specified. The four symbols appearing inside vertical bars `|` are the initial states of the new cells produced by the fission. For example, `f;*;*;*;*;|a;b;c;d|` says that a cell in state `f` will always fission into four cells where the upper-left cell has state `a`, upper-right cell has state `b`, lower-left cell has state `c`, and lower-right cell has state `d`.

There are several shorthands we use in rules. {<*symbol*>;$^+$} denotes a set of symbols with at least one member. When used as a neighbor state, any combination of those symbols can appear as the neighbor states. [<*symbol*>;$^+$] denotes an ordered sequence of symbols with at least one member and can only appear in rules where at least the initial and result symbol are ordered sequences with the same cardinality. Neighbors can be ordered in rules of this form but must specify a sequence with the same cardinality as the initial and result sequence. For example, ordered rule `[x;y];x;*;[a;b];*;[y;x]` is exactly equivalent to the rules `x;x;*;a;*;y` and `y;x;*;b;*;x`. Note that the expected state to the north, `x`, appears in both rules but `a` to the south only appears in the rule starting in state `x` (not `y`). The main reason this is used is in conjunction with the interpretation of <*nonterminal*> to be discussed shortly to state large sets of rules in a concise way.

The set of <*alias*> symbols depicted as atomic above are actually aliases for sets of elements selected from the symbols produced by <*literal*>. A new <*alias*> is created using the following format:

$<alias> = \{(<symbol>;)^{+}\}$.

Before any transitions are done by the MRCA, each instance of $<alias>$ is replaced with the specified list of symbols. All instances of $<alias>$ must be defined before they are used. For example, `allDashes={-;0l-;1l-;2l-}` means that all instances of *allDashes* are replaced by the four dash types given here.

4.2 Computing Sets with the MRCA Simulator

One use of the simulator described in Section 4.1 is to implement the construction described in the proof of Lemma 3.3.1 in Section 3.3. First we will examine the requirements on the one-dimensional CA that is transformed into a computational unit and then examine how additional changes to integrate with an MRCA can be accomplished.

4.2.1 Requirements on the Input CA

Recall computable cellular k-coloring $\psi : \mathcal{Q} \dashrightarrow [k]$ for $X \subseteq [0,1]^2$ and that there exists a Turing machine (TM) M that identifies the correct color of that square with respect to ψ (if it exists) given an encoding of a dyadic-width square $Q \in \mathcal{Q}$. The translation of such a TM to a CA is direct and simple — the state of the TM and location of the head can be encoded into the states of the CA and the transitions can be written to only change the cell at the head. Since TM transitions consult only cell state and machine state, this is sufficient to create an equivalent CA.

Given this transformation, what should this input Turing machine M do precisely? We will examine several concerns and give an example way to address these concerns.

(A) Given a reserved state `GREEN` (`RED`) to denote $\psi(Q) = 1$ ($\psi(Q) = 0$, M must change the entire tape to this state if it concludes that $\psi(Q)$ is 1 (0), respectively.

(B) The address information must be preserved or locally recoverable for the length of the computation. This information must be found in a simply-specified location — we use the start of the machine up to a special symbol `C`.

(C) Add an extra cell to the left end of the CA. This reserves space to assist in coloring and provides a buffer between units of computation. Call this the "rabbit hole" [8].

(A) is easily accomplished by removing the final state(s) of the Turing machine computing ψ and replacing them with states and rules such that two "coloring" states `cGREEN` and `cRED` change all cells to the left edge of the tape with the computed color then start changing all tape cells to the right that color. This process is an infinite process as the tape is right-infinite. When translated to a computational unit in an MRCA, this infinite process is truncated at the right end of the computational unit. When a cell becomes `cGREEN` or `cRED`, it will always change to `GREEN` and `RED` as its next step. The intermediate states `cGREEN` and `cRED` exist to allow correct interaction with the rest of the MRCA as described later.

Many Turing machines for computing a problem do not preserve their input or any way to recover that input. However, if such a TM is provided, we can easily alter that Turing machine by inserting a pre-process that copies the address to the right of the computational area and reserves that address information for the length of the computation in order to address (B). Alternatively, some computations (such as those used in our example in Section 4.3) only change the input a small amount so we can add a post-process to correct the address before it is used elsewhere.

(C) an easy alteration — just add such a cell to the left. We use a cell in state `h`.

4.2.2 Changes to Generate A Computational Unit

The following are the changes to translate the input CA into the set of rules we call a computational unit C_M in our proof of Lemma 3.3.1 . The proof shows that these changes are sufficient so we will now concern ourselves with implementing these changes on an input CA.

(1) Lay out the cells such that each cell is twice the length and width of the cell to its left and half the length and width of the cell to its right.

(2) Slow computation by a factor of three. This is to allow time for periodic fission of new cells to the right of the computation. This is implemented for cells unaffected by this fission as follows: if the original computation had the rule $\delta(a, b) = c$ for some $a, c \in S$ and $c \in N_S$, add new rules $\forall d \in N_S$ $\delta(a, b) = c'$, $\delta(c', d) = c''$, and $\delta(c'', d) = c$.

(3) Add rules to allow for both periodic and computation-prompted fission.

(4) Add rules to pass a copy of the address information off the right end of computation.

(1) is fairly simple. When the CA is translated to a ruleset for C_M, each transition ignores the cells to its north and south. However, each transition reading a state to its west now requires that state to exist only in index $b = 0$ to the west (i.e., the lower cell) because its west neighbor will be half its size. We then follow the description given in **(1)** to laying out the initial cells.

(2) can be implemented as given above and then altered as needed for steps **(3)** and **(4)**. Periodic fission as required by **(3)** can be accomplished by reserving a space on the right end of C_M that represents counting down a set amount and then sending a signal to fission to the left. For example, it could start in a state named f_{100} and subtract one from the index of f before fissioning at f_0. The cell to the west of the fissioning cell would be able to recognize this fission and fission itself — this process repeats down the computational unit. Likewise, when computation requires more space, the cell with the countdown state f_i would be overwritten by the incoming information and fission, creating a new cell with state f_{100} in the appropriate eastern cell.

As a practical matter, it is easier to pause computation until the information in C_M can be pushed all the way to the west, up against the rabbit hole. In our example in Section 4.3, we created the suffix `w` (for “wait”) that indicated that the cell was not to be used in current computation but instead be passed to the west. When a "wait"ing state reads a non-waiting or rabbit hole cell to its west, it begins reading its neighbors according to the instructions in the C_M set of rules and performing computation again.

The pauses implemented for **(2)** are used here to give C_M time to recognize that a fission is going on and reduce the number of possible states that may fission as they read a fission to their east. However, the pauses are only crucial in implementing **(5)**.

The simplest way to accomplish **(5)** is to simply effect the address passing before any computation is done. This will ensure that, regardless of whether C_M halts, the address information will be passed to the east and then north (relative to the rotation of C_M) for seeding of a child set of computations as described in the next section. Alternatively, one can take advantage of the pauses instituted by **(3)** to pass along address information after some point in the computation.

4.2.3 Additional Rules to Complete the Construction

After creating the rules for C_M, the MRCA constructed, A, as given in the proof of Lemma 3.3.1 requires some additional rules to color the space computed by a C_M and initialize new sets of C_M for ever-smaller dyadic squares. These rules can be reused for different computational units as long as cell state names in C_M are understood. For example, if each C_M changes to cells in state `GREEN` or `RED`, A will respond in the same way regardless of what C_M was computing.

First, note that the rule set given in C_M must be rotated 3 times to generate the "pinwheels" depicted in Figure 3.6. A's first set of rules serve to create the initial pinwheel with $C_M(Q(0,0,0))$ in all four directions. As a practical note, $C_M(Q(0,0,0))$ will never actually change to a particular color (this would correspond to the entire unit square being a single color) so it seems wiser to start off with four pinwheels, one each for $C_M(Q(1,0.0,0.0))$,$C_M(Q(1,0.1,0.0))$, $C_M(Q(1,0.0,0.1))$, and $C_M(Q(1,0.1,0.1))$. This also simplifies address specification as all addresses can be written as binary fractions without a whole number part.

This process of initialization also creates a large amount of unused space. Many of A's transitions besides those of C_M and its three rotations will be dedicated to telling these blank spaces to remain blank. Also to manage is the fact that, as C_M fissions for more computational space horizontally, more blank space will be created vertically.

As each C_M computes, there are two points where the rules for A must interact with a C_M. First, if the entire C_M changes to a color `GREEN` or `RED`, A must have a set of rules to accomplish secondary coloring described in detail in Section 3.3. The simplest way to accomplish this is to have C_M use intermediate states `cGREEN` and `cRED` to indicate that cells in C_M should change to `cGREEN` and `cRED` as far to the east and west as possible, up to and including the rabbit hole or the last cookie state to the west (respectively). This is the primary coloring process that, due to the way that computational units are laid out by A, will color only C_M. Secondary coloring occurs when two neighbors, one vertical and one horizontal, are both `GREEN` or `RED`. The cell can then change to this color using the rules specified by A.

If we did not institute intermediate states `cGREEN` and `cRED` for coloring the space used by $C_M(Q(n, i, j))$ and attendant cookie states, the neighboring $C_M(Q(n, i+1, j))$ will react as if it were in the primary coloring stage as we would be unable to differentiate between cells colored by the primary and secondary methods. By having this intermediate state, neighboring C_M cells ignore the presence of `GREEN` and `RED`, reacting only to `cGREEN` and `cRED`. Note that there is no circumstance where cells in a C_M will be colored by the secondary coloring process.

The other central task of A is propogating $C_M(Q)$ to a new pinwheel computing $C_M(Q')$ where Q' has one higher level of precision and the x and x address information is appropriately updated to reflect the new dyadic square being computed. This is also the only case where the rotation of the C_M matters as each value of i and j in dyadic square $Q(n, i, j)$ will be incremented (or not) based on the rotation of C_M (as shown in Figure 3.8).

This is the most complicated part of A as the pinwheel must be centered on the center of the dyadic cell each C_M is computing. Initially marking where the center of the reserved space for each C_M was helped seeding the child pinwheel as it also marked the horizontal center of the child dyadic cell above each C_M (before rotation). The vertical center of the dyadic cell above each C_M could be deduced in each case based on the fission pattern of the empty cells above C_M and the rotated C_M to the west that is computing south-to-north.

Once the center is determined, the address information must be taken from the parent C_M, passed to the right, upwards, and then duplicated to the east, west, and north (before rotation). We found that it was easiest to add an extra bit to the x and y addresses (0 or 1 depending on rotation) at this point. Once the address information is sent, transitions start the use of the rules for the computational unit C_M.

4.2.4 Simplifications for Halting Input CAs

If we know that the input CA will halt*, a few simplifications can be used that provide advantages in showing how the MRCA construction works. For this reason, we chose to use these simplifications in our example (Section 4.3) but they can be applied to any halting CA.

(1) All cells in C_M can be of the same size.

(2) Passing of address information to create child pinwheels can be done after C_M is complete.

(3) There is no need for a pause between steps of C_M.

The justification for this simplifications is that, when a C_M is known to halt, we do not need to seed a new pinwheel of computation while that C_M is still computing. In the

*Note that, under most definitions, a CA never halts but simply keeps applying the transition rules. In our case, we consider a CA halted if the tape consists solely of `GREEN`, `RED`, or blank cells.

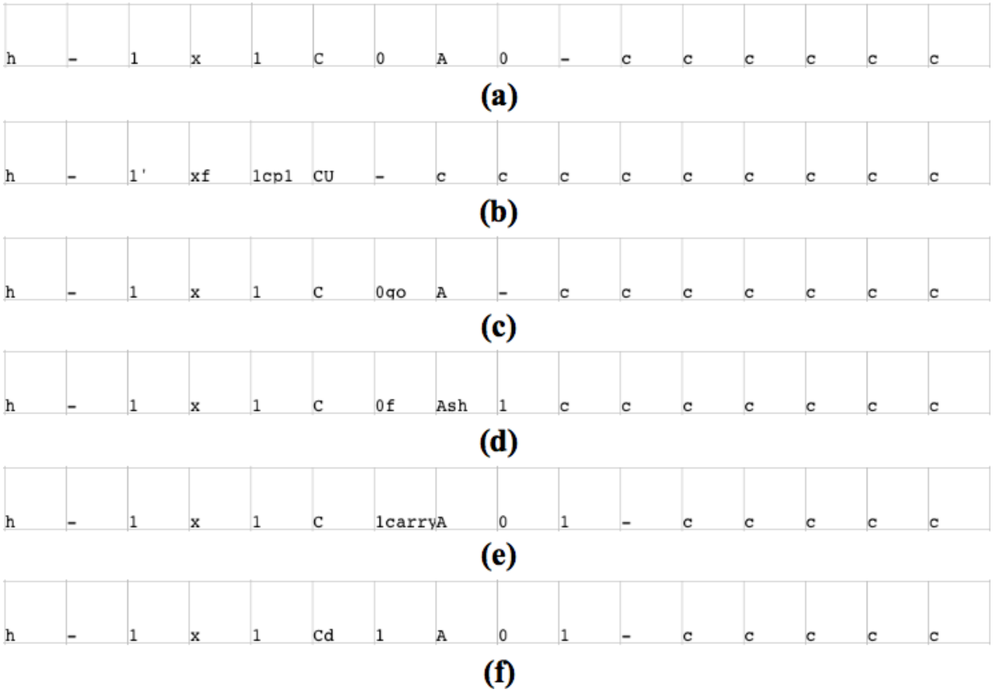
general case, this is a necessary step because the child computations must eventually be seeded in order to guarantee coloring of all dyadic squares when the parent does not halt. If $C_M(Q(n, i, j))$ were to run forever, all dyadic squares with $i * 2^{-n}$ and $j * 2^{-n}$ as prefixes of their x and y coordinates would never be computed unless the address information was sent on.

This possibility of child computations and parent computations occurring at the same time is what necessitates having each cell of computation in C_M be a different size. As the computation progresses, the computational unit is compressed to the west to get out of the way of child computations (and grandchild computations and etc.).

However, if $C_M(Q)$ is known to halt, we do not need to worry about child computations not being initialized. We can keep all cells in the computation the same size and wait to create the child computations until after $C_M(Q)$ halts. This also means that we do not need to pause between computational unit steps because there is no task we need to accomplish while $C_M(Q)$ is running.

This derives some secondary benefits. First, when using the simulator shown in Section 4.1, we, as humans developing the rule sets, can see the states of all cells in a computational unit at one time. This is a problem in the general case because a computational unit of length n with a rabbit hole (i.e., smallest cell) of height s has a last, east-most cell of size $s * 2^n$. For most n, this is a problem as we can't zoom to view cells near the rabbit hole while also comfortably reading the cells to the east. Thus, while the MRCA has no issues with cells doubling in size to the east, human rule development is difficult.

Also, because the address passing and computational unit steps can be separated (as opposed to occurring in parallel some times), it is easier to debug rules created for computation and rules created for creation of child computational units. Although some of the advantages of the parallel nature of the MRCA is lost, the gain in development and presentation simplicity is substantial.



h	-	1	x	1	C	0	A	0	-	c	c	c	c	c	c

(a)

h	-	1'	xf	1cp1	CU	-	c	c	c	c	c	c	c	c	c

(b)

h	-	1	x	1	C	0qo	A	-	c	c	c	c	c	c	c

(c)

h	-	1	x	1	C	0f	Ash	1	c	c	c	c	c	c	c

(d)

h	-	1	x	1	C	1carry	A	0	1	-	c	c	c	c	c

(e)

h	-	1	x	1	Cd	1	A	0	1	-	c	c	c	c	c

(f)

Figure 4.2 Example states of a one-dimensional CA for $\{(x, y) \in [0, 1]^2 \mid y < x^2\}$ generating the value of x^2.

4.3 An Example of MRCA Computation

We will now discuss the specific example of computing $L = \{(x, y) \in [0, 1]^2 \mid y < x^2\}$. This language is computable by a CA that always halts so we will use the simplifications discussed in Section 4.2.4. A complete rule file can be found in Appendix B including the symbols for the $<$*alias*$>$ and $<$*literal*$>$ productions (differentiated by the fact that the former must be defined before use as discussed in the last section).

4.3.1 Input One-dimensional CA

The CA for $L = \{(x, y) \in [0, 1]^2 \mid y < x^2\}$ proceeds through the following steps for any input $Q(n, i, j)$. We first discuss how we generate the value of X^2 in an accumulator (depicted in Figure 4.2). All figures in this section are screenshots of the graphical interface for the simulator presented in Section 4.1.

(a) The CA is initialized to `h-XxYC0A0` where X and Y are the bits representing $X = i * 2^n$ and $Y = j * 2^n$. The bits between the `C` and `A` will represent the "counter" and the rest represents an "accumulator" in which we will calculate X^2. `h` is the rabbit hole state while the blank `-` to the east of it is to store tentative coloring information as discussed later in this section. Figure 4.2(a) shows the initial state for $Q(1, 1, 1)$.

(b) The first task is to set the counter for calculating X^2 by copying the bit(s) of X into the counter section. Figure 4.2(b) shows the process of copying over the value of X, `1`, in descending order where state `CU` represents that the counter has not been set, `1'` represents that this bit has been copied to the right, and `1cp1` represents that the value of the cell is `1` (in this case, the value of Y) and the value being passed to the east is `1`.

(c) When the counter value is set (detected by passing over `CU`), we decrement the least bit of the counter and go to retrieve a copy of X. If this counter results in turning the entire counter to 1s, we go to step **(f)** to continue computation instead. However, in most cases, a "go" signal is sent west to retrieve another copy of the value of X as shown in Figure 4.2(c).

(d) The same procedure used to copy over the counter value in part **(b)** is used here to copy the value of X over to the accumulator section. The bits are again sent in descending significance order. Figure 4.2(d) depicts the CA's state after the value of `1` has been set as the initial accumulator value (the Ash value is explained in the next section).

(e) So far, we have accomplished the multiplication $1 * i$ for $Q(n, i, j)$ and stored that result. However, we want to end up with the value $(i * i) * 2^{-n}$ so we have to "pad" the result of the multiplication with n bits to represent the fact that the X being squared is actually a binary fraction, not a binary whole number. This is accomplished by sending a "shift" `sh`-suffix signal to the left of the counter and copying one `pad` bit per bit of the counter. As the counter has the same length as X and X will always have n bits by construction, this correctly pads our accumulator. Figure 4.2(e) shows the state of the CA after this padding is complete.

(f) After this, the counter is decremented and, if not all 1s, another copy of X is copied to the right and added to what is in the accumulator (from most significant bit to least). Figure 4.2(f) depicts the situation after this where the suffix of d indicates that we are sending a signal that we are done with multiplication and ready to compare Y and X^2.

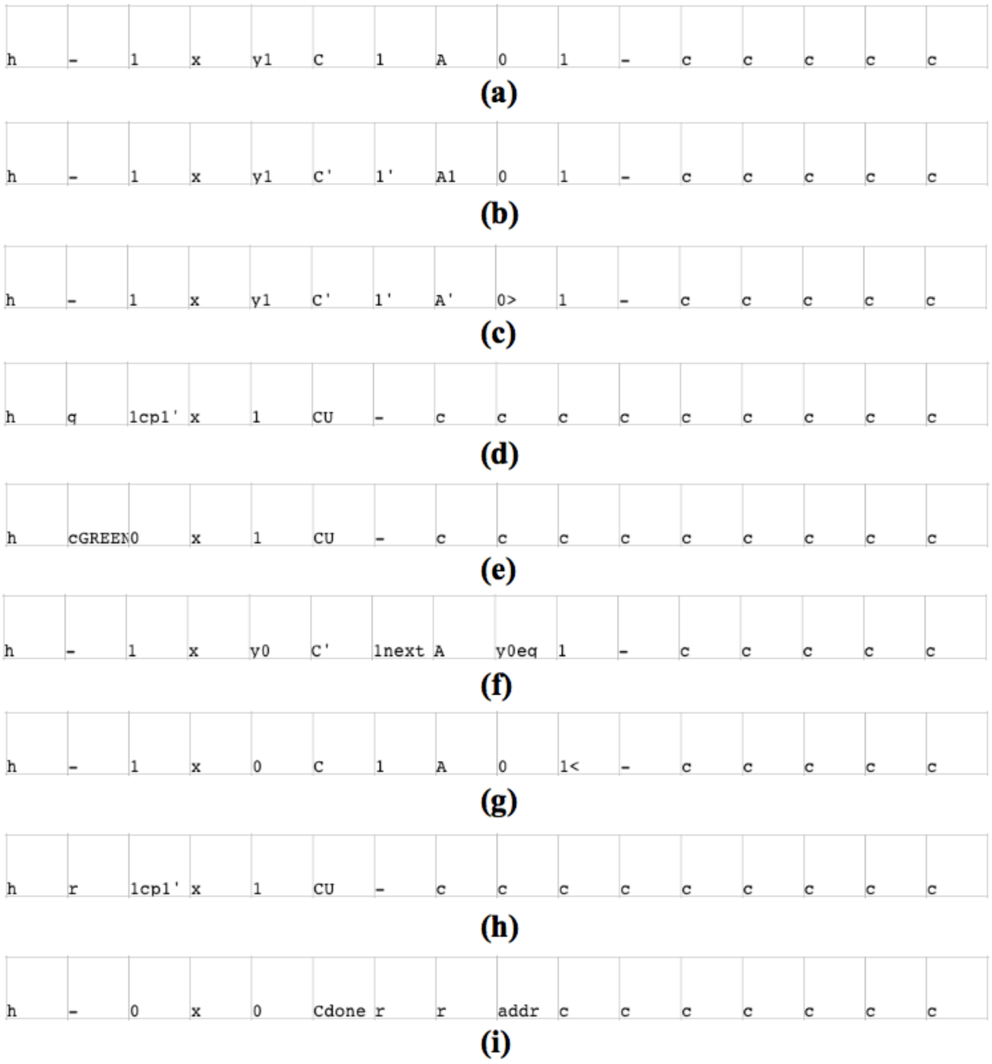
The next series of transitions concern comparing the value of Y to X^2 (as now appearing in the accumulator). Select stages of different computations are depicted in Figure 4.3.

(a) Figure 4.3(a) shows the state of the CA after the d suffix has been passed west until encountering the x symbol in $C_M(Q(1,1,1))$. This means that the current d-suffixed cell is the most significant bit of Y and should be compared to the most significant bit of X^2. So we mark this cell y1.

(b) The value of 1 is passed to the right until encountering the end of the counter marker, A, which becomes C1 as depicted in Figure 4.3(b).

(c) A comparison is now made. If the first bit of X^2 is less than the first bit of Y, we generate the state 0> as shown in Figure 4.3(c). If we had instead been copying over a value of 0 and the first bit of X^2 was 1, we would generate the state 1<. Lastly, if the bit of Y and the bit of X^2 match, a state y0eq or y1eq is generated and a request is sent west to retrieve the next bit of Y. These first three stages would then repeat until a 0> or 1< result occurs or $Y = X^2$ is detected. We will explore examples where each of these results occur.

(d) To continue with the example of 0> for a moment, $Y > X^2$ means that the point $(i * 2^{-n}, j * 2^{-n}) \in L$. Although we are interested in the entire dyadic square $Q(n, i, j)$,



h	-	1	x	y1	C	1	A	0	1	-	c	c	c	c	c

(a)

h	-	1	x	y1	C'	1'	A1	0	1	-	c	c	c	c	c

(b)

h	-	1	x	y1	C'	1'	A'	0>	1	-	c	c	c	c	c

(c)

h	q	1cp1'	x	1	CU	-	c	c	c	c	c	c	c	c	c

(d)

h	cGREEN	0	x	1	CU	-	c	c	c	c	c	c	c	c	c

(e)

h	-	1	x	y0	C'	1next	A	y0eq	1	-	c	c	c	c	c

(f)

h	-	1	x	0	C	1	A	0	1<	-	c	c	c	c	c

(g)

h	r	1cp1'	x	1	CU	-	c	c	c	c	c	c	c	c	c

(h)

h	-	0	x	0	Cdone	r	r	addr	c	c	c	c	c	c	c

(i)

Figure 4.3 Example states of a one-dimensional CA for $L = \{(x, y) \in [0, 1]^2 \mid y < x^2\}$ while performing comparisons. (a)-(c) are of $C_M(Q(1, 1, 1))$, (d)-(e) are of $C_M(Q(1, 0, 1))$, (f)-(h) are of $C_M(Q(1, 1, 0))$, and (j) is of $C_M(Q(1, 0, 0))$.

this is useful information. So we tentatively guess that this dyadic square should be green — in the very least, it should not be red! In the process of marking the square to the right of the rabbit hole `h` with a state `g` to indicate tentative green-ness, X will have it's lowest bit increased because we now are going to check if the point $((i+1)*2^{-n}, j*2^{-n}) \in L$ is also green. If it is, it is simple to see that the entire dyadic square $Q(n,i,j) \subseteq L$. We switch examples now to see this situation in Figure 4.3(d) which shows the processing of $Q(1,0,1)$ after the "guess" of green g has been recorded and the CA is starting to generate X^2 as per the multiplication rules, part (a)[†].

(e) If multiplication and comparison results in the finding that $Y > X^2$ for this updated value of X, we now begin the coloring process for this CA. Figure 4.3(e) shows how our tentative guess state g becomes state $cGREEN$ and starts the coloring process of this CA. Note that X is fixed to send the correct address of the lower-left corner of the dyadic square in case the result had not been a coloring.

(f) Figure 4.3(f) shows the example computation of $Q(1,1,0)$ where we have compared the first (and only) bit of Y, `0`, to the first bit of the accumulator and found them equal. This creates the "next" series of suffix states that are sent west to retrieve the next bit of Y.

(g) When the next bit of Y does not exist, this does not neccessarily mean that $X^2 > Y$ as the remaining bits of the accumulator may be `0`. Figure 4.3(g) shows that this is not the case — the CA passes a signal to the right to verify that all the remaining bits of X^2 is `0`. As this is not the case, we create the signal `1<`.

(h) When the signal `1<` reaches the rabbit hole state, we record an `r` state to indicate both that $(i*2^{-n}, j*2^{-n}) \notin L$ and we tentatively believe that $Q(n,i,j)$ is red. We now

[†] We switch from $C_M(Q(1,1,1))$ because it enters a special case where increasing X's least significant bit results in $X = 1$. This always means that $Y < X^2$ so we save some computational steps by immediately reacting as if $<$ were sent west.

verify that $(i * 2^{-n}, (j+1) * 2^{-n}) \notin L$ — if this is also the case, we can safely conclude that $Q(n, i, j) \nsubseteq L$. Figure 4.3(h) depicts the situation where the computation of this second point is just starting.

(i) Figure 4.3(i) depicts an instance of the final group of situations where $Y = X^2$ (in this case, for $Q(1, 0, 0)$. In this situation, we can never color Q red or green so computation concludes.

Through a combination of the processes for multiplying to generate X^2 and the process to compare that value to Y, we compute whether two points in each dyadic square is in L and consequently decide if the computational space should be colored and, if so, whether it should be colored green or red. We discuss how fission for computational space is handled in Section 4.3.3 as the simplest reason to fission for space occurs during child computational unit pinwheel creations. When input into our construction, all that remains to rotate the rules to create pinwheels, write rules to color the entire dyadic square, write rules to pass along updated addresses, and write rules to create child pinwheels with those new addresses.

4.3.2 Rotation and Coloring

Now we can discuss how the MRCA A is initialized using the CA discussed in the previous section.

First, all rules written for the CA must be translated to MRCA rules. Fortunately, due to our simplifying assumptions we just need to add $*$s as the north and south neighborhood state positions for all one-dimensional rules. Second, we have to create an alternate rule set for each of the 3 other wheels of the computational unit pinwheel. In our case, we use the prefix $1l$, $0l$, and $2l$ for the north-to-south, east-to-west, and south-to-north computational units as depicted in Figure 4.4. The rotations of the rule sets are simple — for example, to generate the north-to-south rules, we simply move all neighborhoods one step clockwise (e.g., east neighbor specifications become north neighbor specifications) and add a prefix. For

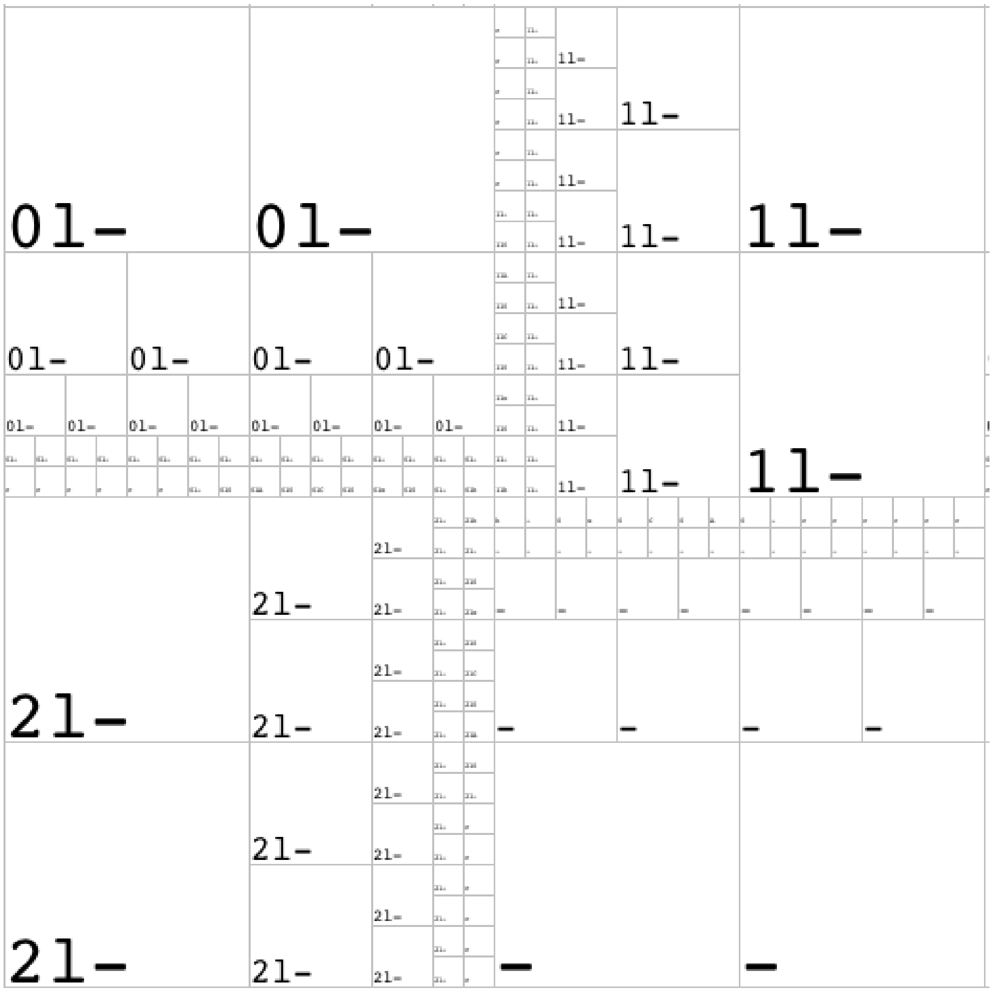
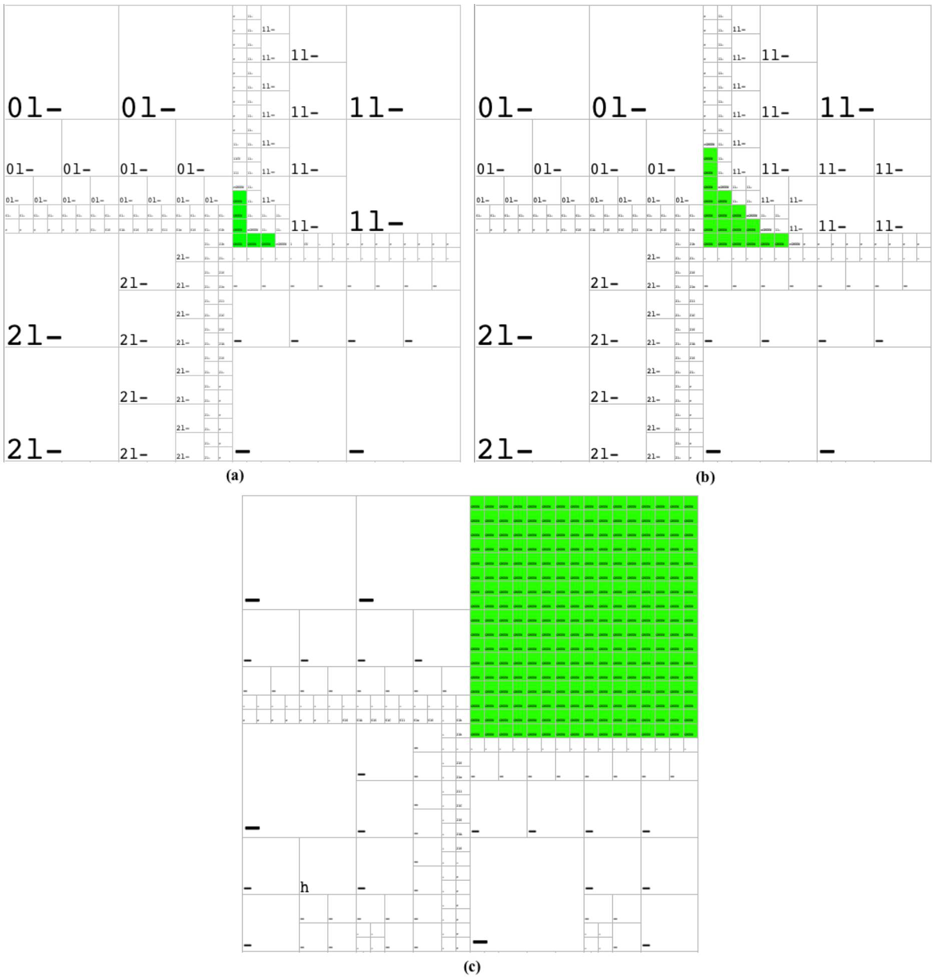


Figure 4.4 Initial states of an MRCA for $\{(x,y) \in [0,1]^2 \mid y < x^2\}$ depicting only the pinwheel for dyadic square $Q(1,0,0)$.



(a)

(b)

(c)

Figure 4.5 Configurations of an MRCA for $L = \{(x, y) \in [0, 1]^2 \mid y < x^2\}$ depicting a coloring process for dyadic square $Q(1, 0, 1)$.

example, the rule $x; a; b; c; d; y$ becomes the rule $1lx; 1lb; 1lc; 1ld; 1la; 1ly$ for south-to-north computation. Note that even the blank spaces in Figure 4.4 have the respective prefixes.

After the computation of each computational unit is complete, the computational unit may be a color. Because the same input is given to each unit in a pinwheel, we will have each unit changing to the same color. This is shown (with directions $2l$ and $0l$ frozen) in Figure 4.5(a). Then secondary coloring, in this case if a cell's south and west neighbors are green, occurs (Figure 4.5(b)). Figure 4.5(c) shows the completed coloring — note that, due to how the primary coloring process of computational units is accomplished, the color

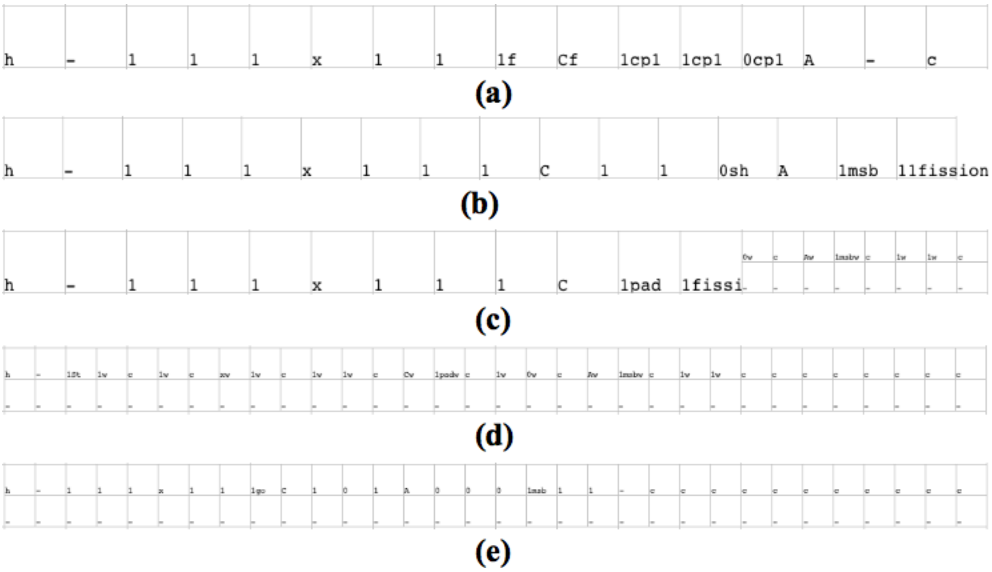

h - 1 1 1 x 1 1 1f Cf 1cp1 1cp1 0cp1 A - c
(a)
h - 1 1 1 x 1 1 1 C 1 1 0sh A 1msb 11fission
(b)
h - 1 1 1 x 1 1 1 C 1pad 1fissi
0v c Av 1msbv c 1v 1v c
(c)
h - 10t 1v c 1v c xv 1v c 1v 1v c Cv 1padv c 1v 0v c Av 1msbv c 1v 1v c c c c c c c c
(d)
h - 1 1 1 x 1 1 1go C 1 0 1 A 0 0 0 1msb 1 1 - c c c c c c c c c c c
(e)

Figure 4.6 Example of fission due to computational concerns when computing $Q(3, 111_{binary}, 111_{binary})$. A single computational unit is shown.

will not "bleed" into neighboring dyadic square computation. Also note that west-to-east computation is part of the fourth quadrant but, as the entire square will eventually be colored, exactly when each computational unit changes to that color is of little consequence.

4.3.3 Fission and Creation of Child Pinwheels of Computational Units

The final areas to discuss include when fission is required and how to create child pinwheels of computational units. When these issues are addressed, our example MRCA constructed for computation of L will be complete!

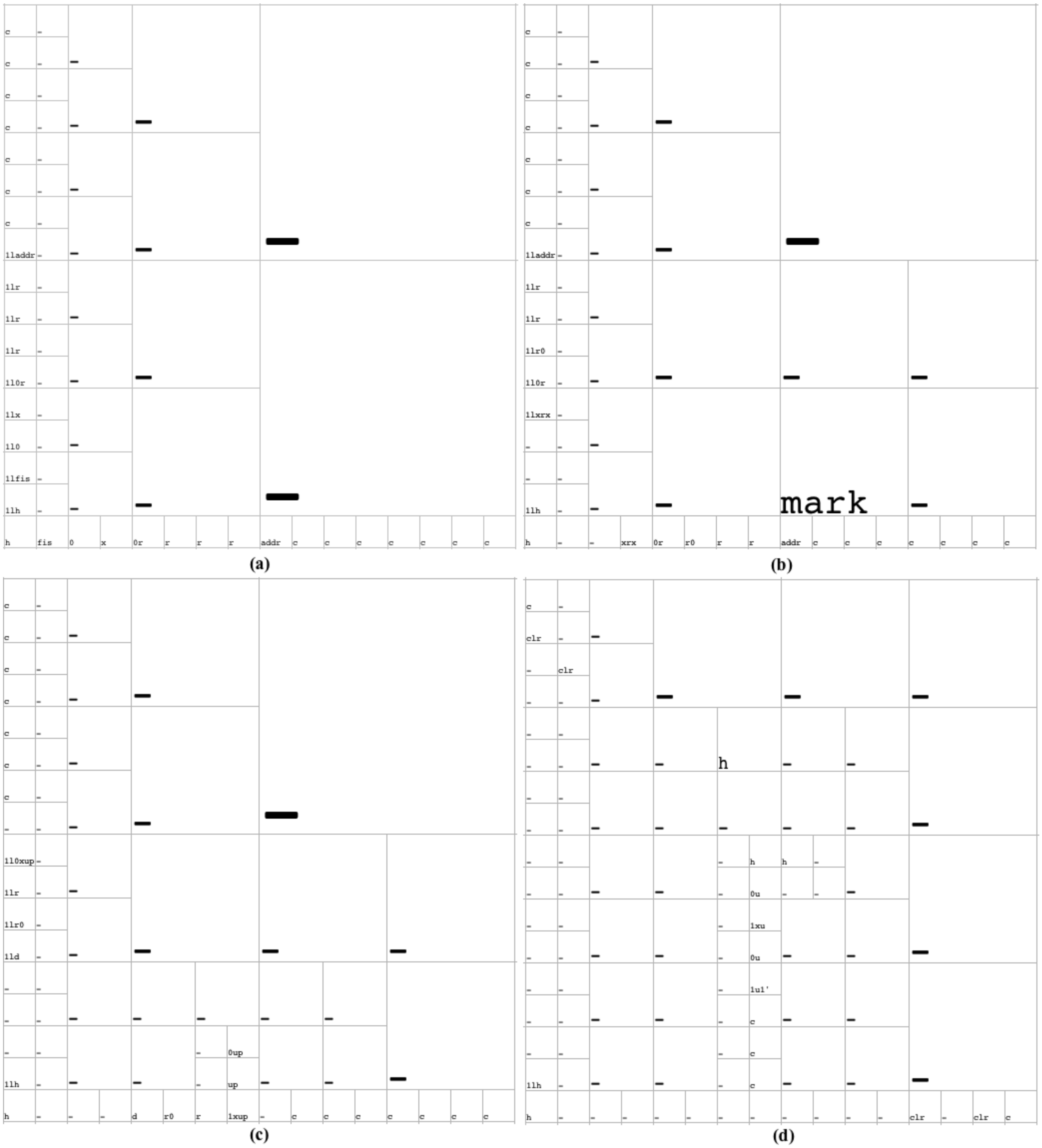
Before continuing, note that, although it did not occur in the pictured computations for L, it is possible that computation will require more space than that allotted to the unit initially. In the CA, we assumed that there was an infinite number of cells (marked c for cookie states) to the right. In the MRCA, we do not have this luxury so we must alter the rules to allow fission.

Figure 4.6 shows a situation where, during the processing of $Q(3, 111_{binary}, 111_{binary})$, we must fission for more space. Specifically, in part (a) we are passing the value X of 111 to the right and part (b) depicts the transition where we have attempted to pass a 1 to the right but the cell to the right either does not exist (as shown here) or is of the wrong computational unit prefix. Part (c) shows an intermediate stage where the computation of the CA has continued while the right hand side of the CA has frozen to fission (using the suffix `w` for "wait"). Fortunately, because fission for more computational space can only occur during this process of accumulating X^2, we can write rules to handle all possible situations of collision between the left hand side continuing computation while the right is fissioning. Figure 4.6(d) shows the last step of the fission. As the fission continued to the left, non-cookie wait symbols are also passed to the left and the leftmost bit in the computational unit has changed to the `St` suffix to indicate that computation should "start" again as this suffix is passed east. Figure 4.6(e) shows a configuration after fission is complete showing computation continuing.

The other situation where fission is required is as part of the creation of child pinwheels of computational units. Selected configurations for creating the child pinwheel of $Q(1, 0, 0)$ above the east-to-west computational unit (i.e., the computational units for $Q(2, 01_{binary}, 01_{binary})$) are depicted in Figure 4.7.

(a) Figure 4.7(a) depicts the first step, where the space allocated to the parent computational unit has been partially cleared out with cells in state `r`, a cell at the horizontal center of the child pinwheel is in state `addr`, and the space reserved next to the rabbit hole has changed to state `fis` to start operations to create a new set of computational units.

(b) From this point, two series of transitions occur in parallel. First, the `fis` state is passed up and over until we read the `addr` state below or the edge of the dyadic square is detected. The first state to do this changes to a state `mark` and the cell to its left becomes states `ar` and `a` to indicate where address information will be sent up. Second,


c
c
c
c
c
c
c
1laddr
1lr
1lr
1lr
1l0r
1lx
1l0
1lfis
1lh
h
fis
0
x
0r
r
r
r
addr
c
c
c
c
c
c
c
c
c
c
c
c
c
c
1laddr
1lr
1lr
1lr0
1l0r
1lxrx
1lh
h
xrx
0r
r0
r
r
addr
c
c
c
c
c
c
c
mark
c
c
c
c
c
c
c
1l0xup
1lr
1lr0
1ld
1lh
0up
up
h
d
r0
r
1xup
c
c
c
c
c
c
c
c
clr
clr
h
h
h
0u
1xu
0u
1u1'
c
c
c
1lh
h
clr
clr
c

(a)

(b)

(c)

(d)

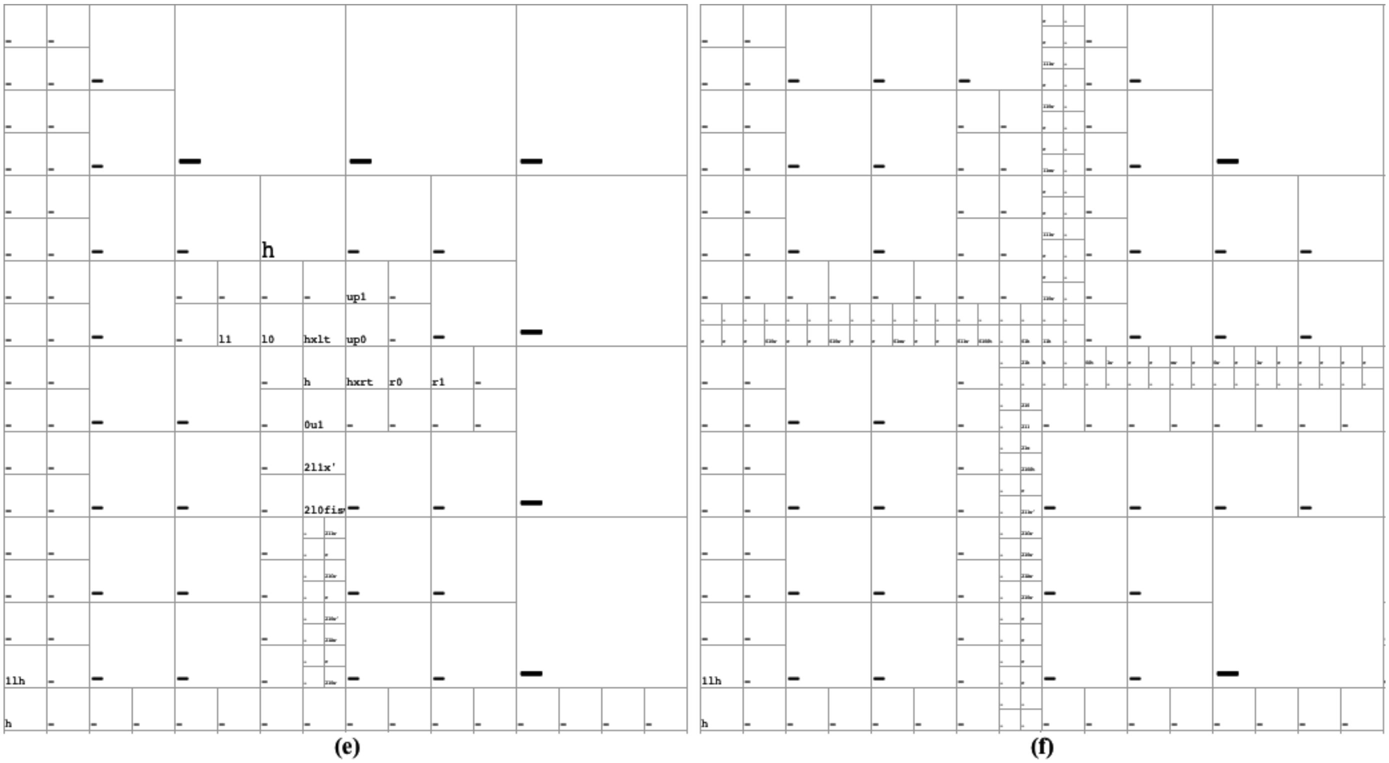


Figure 4.7 Example of fission due to computational concerns when computing $Q(1, 0, 0)$. Note that computation continues in the $1l$ rotation of this unit.

address information is sent to the east. Figure 4.7(b) shows the configuration after the first series of transitions is complete.

(c) Figure 4.7(c) shows how the address information is passed to the north with states suffixed by `up`. Blank cells to the north fission in order to accept this information. Note that a `1` has been added to the least significant end of the value of X (this is also done with the value of Y before the next subfigure).

(d) The center of the new pinwheel is found through the pattern of fissions that the empty cells must go through and that sometimes results in extra empty cells changing to state `h` (along with the correct space). Figure 4.7(d) depicts a configuration after we have sent address information up to the center of the child pinwheel marked by `h`. Our next step is to copy the last bit of the vertical computational unit up and through the rabbit holes for each of the other three wheels. This is done using the `u` series of states such as $1u1'$ appearing in this figure which represents remembering the address `1` on the left of the symbol but passing the `1'` on the right of the symbol.

(e) When the first bit gets to the rabbit hole for the $2l$ (i.e., north-to-south) computational unit, that bit is passed through the rabbit hole as shown in Figure 4.7(e). Note that, as bits are passed to the north through the center of the pinwheel, they change to computational unit states like $2l0$, $2l1$, $2lC$, etc. and, as computation starts at the end (south in this rotation), all the states will have been sent and computation can proceed. Note that there is a slight delay to sending the address information such that the north computational unit will be a few steps behind the east and west computational unit which will be behind the south computational unit. Also, the basic address and computational information will not fit in the southern computational unit so a fission is occurring to the south. This procedure for fission occurs in the same way as described earlier in this section when fission was done for computational reasons.

(f) Figure 4.7(f) shows the completed child pinwheel. Note that fission in the southern computational unit is passed automatically through the rabbit holes. This is because we know that all the other computational units will need to fission just to fit the initial configuration of $C_M(Q(2,1,1))$ so this signal saves computation time and detection rules.

Thus our example MRCA accomplishes the tasks given in the proof of Lemma 3.3.1 and so MRCA computes $L = \{(x,y) \in [0,1]^2 \mid y < x^2\}$ because of the input CA computing this set.

4.4 In-Place MRCA Computation

We now step back from the details of the construction in Lemma 3.3.1 to look at a simpler class of problems for MRCA computation, in-place MRCA computable sets.

Definition 4.4.1. *A set $X \in [0,1]^2$ is* MRCA in-place computable *if there exists a MRCA that MRCA computes X with the restrictions that: each cell fissions at every time step and, for all rules in the MRCA, the state of the current cell is determined solely by the state of the parent cell.*

An MRCA in-place computable set is one where the state of each dyadic square determines the state of a set of child dyadic squares at an additional level of precision. For example, the state of $Q(1,1,0)$ uniquely determines the state of $Q(2, 10_{binary}, 0)$, $Q(2, 10_{binary}, 1)$, $Q(2, 11_{binary}, 0)$, and $Q(2, 11_{binary}, 1)$ in in-place MRCA computation.

A class of in-place computable sets we examine is that of rational polygons.

Definition 4.4.2. *A* rational polygon *is a planar figure that is bounded by a closed path composed of a finite sequence of half-planes delimited by lines with rational slope.*

The problem of creating a set of rules to delineate any rational polygon can be decomposed into creating sets of rules to delineate the border between the polygon and the plane. For

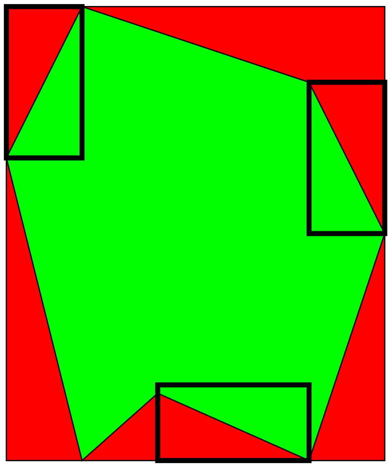


Figure 4.8 Example rational polygon with three identified line boundaries. In the limit of MRCA levels, the accept area is designated in green while the reject area is red.

example, in Figure 4.8, green denotes space inside the polygon, red denotes space outside the polygon, and the black frames decompose the polygon into sub-sections that are either entirely one color or contain at most one line segment. It is always possible to deconstruct a polygon in this fashion.

Thus, if we can build a set of MRCA rules to color portions of the plane red and green for each of these sub-sections, we can merge these rules (much as we did when rotating computational units in the proof of Theorem 3.3.2), using unique marks for each rule set and altering rules to treat cells marked differently as "edge cells" (i.e. state $\perp$). We will now examine the situations that may occur when creating an MRCA for computing regions delimited by a subset of all lines $y = mx + b$ where $m, b \in \mathbb{Q}$.

To create MRCA rules for these lines, we will use the term *rescaled dyadic square*, meaning a dyadic square whose points are addressed as if they were the unit square. We will discuss how a line crosses each square in a configuration as if they were occupying the entire unit square by rescaling the line's x and y intercepts.

To facilitate our discussion, we will refer to the leftmost point of the line inside the rescaled dyadic square of interest as the *entry* point while the rightmost point will be the *exit* point. The squares that have been colored can either be viewed as not fissioning any further or fissioning along with the other cells. In either case, they retain their accept or reject color and represent no more than the simple color-retention computation. Each of the other squares will be addressed by a different subcase (specified by their entry and exit points). We now state our main result for in-place MRCAs:

Theorem 4.4.3. *There exists a finite set of MRCA rules to compute any line* $\ell = \{(x, y) \mid y = mx + b, m, b \in \mathbb{Q}\}$.

The proof of this theorem primarily consists of the observation that any line with rational slope will cross each rescaled dyadic square a finite number of ways. We discover that, for $m = \dfrac{m_n}{m_d}$ and $b = \dfrac{b_n}{b_d}$, the line must enter each rescaled dyadic square at y-axis points $\left\{(0, b \pm \dfrac{i}{2b_d m_d m_n}) \mid i \in \mathbb{N}\right\}$. Given that the slope is constant, there is a finite number of ways that ℓ can cross each square when entering at one of these points.

If we carefully examine the fission of each dyadic square, we can identify analytically the equation of the line in each resulting dyadic square. Because the number of ways ℓ will cross the square is finite, we can write a finite set of rules to identify which case each rescaled dyadic square represents and the four cell resulting from fission: A red square, a green square, or a square that recursively requires the same set of decisions.

The Java code for a graphical user interface to create a set of rules for any part of $[0, 1]^2$ delimited by a rational line is given in Appendix C.

Proof of Theorem 4.4.3. If the line does not enter a given unit square, that square and all dyadic subsquares have no entry or exit point and will be colored by the last ancestor cell containing these points. Therefore we will focus on the dyadic squares that the line crosses.

First, note that all lines of defined, non-zero rational slope cross the y-axis at some value b and cross the x-axis at some value x_{int}. Thus, the slope m is $\dfrac{b}{x_{int}}$ and $(0, b), (x_{int}, 0) \in \ell$.

As m and b are rational, we will identify numerator and denominator pairs $m_n, m_d, d_d \in \mathbb{Z}^+$ and $b_n \in \mathbb{Z}$ such that $m = \dfrac{m_n}{m_d}$ and $b = \dfrac{b_n}{b_d}$.

Second, note that any time the line crosses a dyadic square, the only thing that differs from when we examine any other dyadic square is the y-intercept b and x-intercept x_{int}. When we model a higher resolution (i.e. smaller dyadic squares) including the line, the slope is invariant. Therefore, we will focus on the points at which the line enters and exits a rescaled dyadic square and subsequent (fission) subsquares. Specifically, let us examine the unit square adjacent to the y-axis that contains ℓ. Any other square that ℓ hits is similar but we have to consider a different b and x_{int}.

We now examine nine possible configurations of entry and exit points for such a line to see that they result in lines separated by a minimum of $\dfrac{1}{2b_d m_d m_n}$. We then show these nine configurations are symmetric to all possible configurations. Since there exist rules for each of these nine cases, each case deterministically leads to another, and there is a finite number of ways the line can cross any dyadic square, there is a finite set of rules that specifies the behavior of a MRCA to compute regions delimited by a rational line and our proof will be complete.

The nine cases are specified relative to b, x_{int}, and three other quantities: x_{half}, x_{one}, and y_{half} such that $(\frac{1}{2}, x_{half}), (1, x_{one}), (y_{half}, \frac{1}{2}) \in \ell$. We will label the four *quadrants* of the fissioned version of this rescaled dyadic square as given in Figure 4.9 with A,B,C, and D. Each case identifies a class of entry and exit points for each quadrant.

For example, if we consider rescaled dyadic square A, each point falling inside A but labeled relative to the original unit square would have to be shifted up $\frac{1}{2}$ then have both x and y coordinates doubled to be points relative to A as A is half the length and width of the original square.

There is a finite number of subcases. In each possible case, the entry and exit points are the product of $\dfrac{1}{2b_d m_d m_n}$ and an integer expression. This suffices because there are at

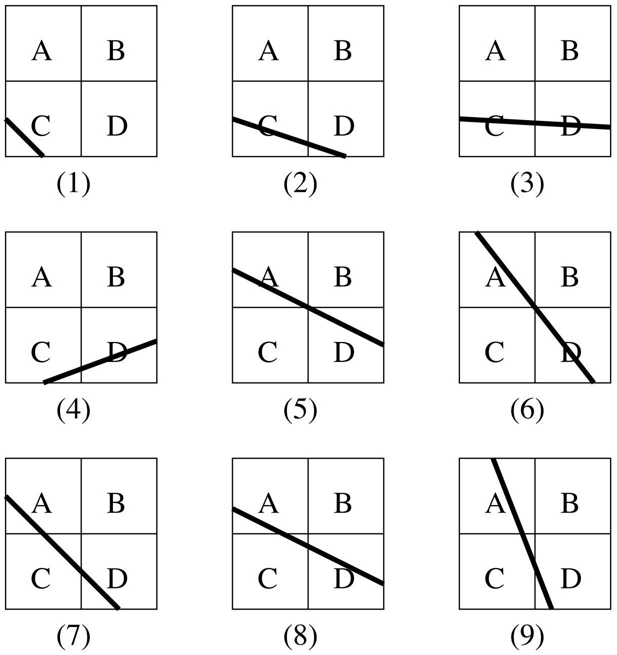


Figure 4.9 Graphical representation of the nine primary ways a rational line can enter and exit a unit square. There are additional but symmetric ways for this to occur.

most $2m$ ways that a line of slope m can cross a unit square while remaining separated by $\frac{1}{2b_d m_d m_n}$.

Note that

$$\begin{aligned}
b = y_{int} &= \frac{b_n}{b_d} \\
&= \frac{2b_n m_d m_n}{2b_d m_d m_n}, \\
x_{int} &= \frac{-b_n}{b_d} * \frac{m_d}{m_n} \\
&= \frac{-2b_n m_d^2}{2b_d m_d m_n}, \\
y_{half} &= \left(\frac{1}{2} - \frac{b_n}{b_d}\right) * \frac{m_d}{m_n} \\
&= \frac{m_d^2(b_d - 2b_n)}{2b_d m_d m_n}, \\
y_{one} &= \left(1 - \frac{b_n}{b_d}\right) * \frac{m_d}{m_n} \\
&= \frac{2m_d^2(b_d - b_n)}{2b_d m_d m_n}, \text{ and} \\
x_{half} &= \frac{1}{2} * \frac{m_n}{m_d} \\
&= \frac{b_d m_n^2 + 2b_n m_d m_n}{2b_d m_d m_n}
\end{aligned}$$

so all of the above notations are integer multiples of $\frac{1}{2b_d m_d m_n}$. We will now examine in detail the nine cases given in Figure 4.9, showing that both entry and exit points are multiples of $\frac{1}{2b_d m_d m_n}$.

(1) C to C. In this case, both the y-intercept b and the x_{int} are less than $\frac{1}{2}$. So we color all quadrants except C identically (i.e. all are either green or red) and examine the line from $(0, 2b)$ to $(2x_{int}, 0)$ in C. Note that $2b$ and $2x_{int}$ can be written with a denominator of $2b_d m_d m_n$ and integer numerator by earlier calculation of b and x_{int}.

(2) C to D with no intermediate quadrant, enter from y-axis, exit by crossing x-axis. We color quadrants A and B identically then examine two subcases: In quadrant C, the line from $(0, 2b)$ to $(1, 2x_{half})$ and, in quadrant D, the line from $(0, 2x_{half})$ to $(2x_{int} - 1, 0)$.

Note that $2x_{half}$ can be written with integer numerator and denominator $2b_d m_d m_n$ and $2x_{int} - 1$ can as well.

(3) C to D with no intermediate quadrant, enter from y-axis, exit by crossing $x = 1$. So we color quadrants A and B identically then examine two subcases: In quadrant C, the line from $(0, 2b)$ to $(1, 2x_{half})$ and, in quadrant D, the line from $(0, 2x_{half})$ to $(1, 2x_{one})$. Note that $2x_{half}$ can be written with integer numerator and denominator $2b_d m_d m_n$ and $2x_{one}$ can as well.

(4) C and D with no intermediate quadrant, enter from x-axis, exit by crossing the line $x = 1$. We color quadrants A and B identically then examine two subcases: In quadrant C, the line from $(2x_{int}, 0)$ to $(1, 2x_{half})$ and, in quadrant D, the line from $(0, 2x_{half})$ to $(1, 2x_{one})$. $2x_{half}$ and $2x_{one}$ can be written as integers over $2b_d m_d m_n$.

(5) A to D with no intermediate quadrant, enter from y-axis, exit by crossing the line $x = 1$. We color quadrants B and C oppositely (i.e. if B is accepted, C is rejected or vice versa depending on the polygon) then examine two subcases: In quadrant A, the line from $(0, 2b - 1)$ to $(1, 0)$ and, in quadrant D, the line from $(0, 1)$ to $(1, 2x_{one})$. $2x_{one}$ and $2b - 1$ can be written as integers over $2b_d m_d m_n$.

(6) A to D with no intermediate quadrant, enter by crossing the line $y = 1$, exit by crossing the x-axis. So we color quadrants B and C oppositely then examine two subcases: In quadrant A, the line from $(2y_{one}, 1)$ to $(1, 0)$ and, in quadrant D, the line from $(1, 0)$ to $(2x_{int} - 1, 0)$. Note that $2y_{one}$ and $2x_{int}$ can be written with integer numerator and denominator of $2b_d m_d m_n$.

(7) A to D with C as intermediate quadrant, enter by crossing the y-axis, exit by crossing x-axis. So we color quadrant B then examine three subcases: In quadrant A, the line from $(0, 2b - 1)$ to $(2y_{half}, 0)$; in quadrant C, the line from $(2y_{half}, 1)$ to $(1, 2x_{half})$; in

quadrant D, the line from $(0, 2x_{half})$ to $(2x_{int} - 1, 0)$. $2y_{half}$ and $2x_{half}$ can be written with integer numerator and denominator of $2b_d m_d m_n$ by earlier calculation of y_{half} and x_{half} and $2b - 1$ can as well.

(8) A to D with C as intermediate quadrant, enter by crossing the y-axis, exit by crossing the line $x = 1$. This differs from (7) only in quadrant D where we instead examine the line from $(0, 2x_{half})$ to $(1, 2x_{one})$. $2x_{half}$ and $2x_{one}$ are integer multiples of $\dfrac{1}{2b_d m_d m_n}$.

(9) A to D with C as intermediate quadrant, enter by crossing the line $y = 1$, exit by crossing x-axis. This differs from (7) only in quadrant A where we instead examine the line from $(2y_{one}, 1)$ to $(2y_{half}, 0)$. $2y_{one}$ and $2y_{half}$ are integer multiples of $\dfrac{1}{2b_d m_d m_n}$.

Last, we show these cases exhaust all possibilities within symmetry. That is, there is a simple transformation of any possible situation to one of the nine cases above. We now enumerate all possible ways that a line can cross into and out of a unit square and show which primary case applies after which simple transformations. Note that translations cannot be applied as part of this process as this may result in failing to preserve the fact that there is at least $\dfrac{1}{2b_d m_d m_n}$ between the x and y coordinate of each point. We now examine all 16 combinations of quadrants the line could cross and show that each combination is matched by a primary after a simple reflection.

- A line not crossing this dyadic square requires only rules to color all quadrants the same.

- A line crossing only quadrant C is case (1): such a line must enter from the y-axis and exit to the x-axis or the line will enter another quadrant.

- A line crossing only quadrant A, B, or D would be reflections of lines crossing quadrants C in case (1). Specifically, reflect over $y = \frac{1}{2}$, $y = 1 - x$, and $x = \frac{1}{2}$ for quadrants A, B, and D.

- A line through quadrants C and D must enter C by crossing the y-axis or the x-axis. It then will exit D by crossing the x-axis or the line $x = 1$.
    - Line entering C by crossing y-axis and exiting D by crossing the x-axis: this is case (2).
    - Line entering C by crossing y-axis and exiting D by crossing $x = 1$: this is case (3).
    - Line entering C by crossing x-axis and exiting D by crossing the x-axis: impossible.
    - Line entering C by crossing x-axis and exiting D by crossing $x = 1$: this is case (4).
- A line through quadrants A and B is symmetric to the case for quadrants C and D when we reflect over the line $y = \frac{1}{2}$.
- A line through quadrants A and D can enter by crossing the y-axis or the line $y = 1$ and exit by crossing the x-axis or the line $x = 1$
    - Line entering A by crossing y-axis and exiting D by crossing the x-axis: must pass through quadrant C unless also exiting by crossing point $(1, 0)$ so case (6) applies.
    - Line entering A by crossing y-axis and exiting D by crossing $x = 1$: this is case (6).
    - Line entering A by crossing $y = 1$ and exiting D by crossing the x-axis: this is case (5).
    - Line entering A by crossing $y = 1$ and exiting D by crossing $x = 1$: must pass through quadrant B unless also entering by crossing the point $(0, 1)$ so case (5) applies.
- A line through quadrants A and C is symmetric to the case for quadrants C and D after reflection over the line $y = x$.

- A line through quadrants B and C is symmetric to the cases for quadrants A and D when reflected over the line $x = \frac{1}{2}$.

- A line through quadrants B and D is symmetric to the cases where we examined quadrants A and C when reflected over the line $x = \frac{1}{2}$.

- A line through quadrants A, C, and D would have to cross through quadrant C as an intermediate quadrant. It would enter A after crossing either the y-axis or the line $y = 1$ and exit D after crossing either the x-axis or the line $x = 1$.

  - Line enters A by crossing the y-axis, exits D by crossing the x-axis: this is case (7).
  - Line enters A by crossing the y-axis, exits D by crossing the line $x = 1$: this is case (8).
  - Line enters A by crossing the line $y = 1$, exits D by crossing the x-axis: this is case (9).
  - Line enters A by crossing the line $y = 1$, exits D by crossing the line $x = 1$: impossible, this line would have to avoid quadrant C somehow.

- A line through quadrants A, B, and C is symmetric to a line crossing A, C, and D after reflection over $y = \frac{1}{2}$.

- A line through quadrants A, B and D would have to cross quadrant B as an intermediate quadrant. It is symmetric to the 4 cases explored in examining lines through quadrants A, C, D if reflected over $y = 1 - x$.

- A line through quadrants B, C, and D would be symmetric to the cases of the line through quadrants A, C, and D when reflected over the line $x = \frac{1}{2}$.

- There is no possible line through all quadrants.

Note that we have omitted lines that satisfy more than one subcase. For example, a line that enters from the point on both the y-axis and the line $y = 1$ or exits by crossing the x-axis and the line $x = 1$. Using either applicable analysis identifies a correct transformation and rules set.

So, by exhaustion, all possible lines are represented (via symmetry) by the nine primary cases we examined. Since there exist rules for each of these nine cases, each case deterministically leads to another, and there is a finite number of ways the line can cross any dyadic square. So there is a finite set of rules that specifies the behavior of a MRCA to compute a line. □

CHAPTER 5. KOLMOGOROV COMPLEXITY AND REGULAR LANGUAGES

Section 5.1 introduces the basics of Kolmogorov complexity necessary to examine regular languages including the Incompressibility Theorem. Section 5.2 explains how the tools of Kolmogorov complexity can be applied to regular languages. Section 5.3 illustrates this approach through use of a simple method of showing languages are nonregular with several examples. Section 5.4 shows the common pumping lemma for regular languages, a language that is nonregular but satisfies the pumping lemma, and less common pumping lemmas that seem to require advanced tools from graph theory to prove.

5.1 Kolmogorov Complexity Results

Before discussing the KC regularity theorem of Li and Vitányi [29], we introduce some basic notation related to regular languages and finite automata. We assume that, in an undergraduate course, these topics will have been discussed in more detail.

Let Σ be a finite nonempty alphabet, and let Q be a finite nonempty set of states. A *transition function* is a function $\delta : \Sigma \times Q \to Q$. We extend δ to $\hat{\delta}$ on Σ^* by $\hat{\delta}(\lambda, q) = q$ and $\hat{\delta}(a_1 \dots a_n, q) = \delta(a_n, \hat{\delta}(a_1 \dots a_{n-1}, q))$. A (deterministic) *finite automaton* (DFA) A is a quintuple $(Q, \Sigma, \delta, s, F)$, where $s \in Q$ is a distinguished *initial state* and $F \subseteq Q$ is a set of *final states*. A language L is any subset of Σ^*. A language accepted by DFA A is the set $L = \{x \mid \hat{\delta}(x, q_0) \in F\}$.

We now discuss some basic results in the area of Kolmogorov complexity assuming knowledge of Turing machines. These topics are not covered in a typical undergraduate course so we include a more complete discussion.

Definition 5.1.1 ([53; 24; 9])**.** *Let M be a Turing machine, and let $x \in \{0,1\}^*$. The* plain Kolmogorov complexity *of x with respect to M is*

$$C_M(x) = \min\{|\pi| \mid \pi \in \{0,1\}^* \text{ and } M(\pi) = x\},$$

where $\min \emptyset = \infty$.

Let C denote the (plain) Kolmogorov complexity function with respect to a universal Turing machine U. We note that universal Turing machines U exist and are optimal: for any machine M, there exists *optimality constant* c_M such that for all $x \in \{0,1\}^*$ $C(x) \leq C_M(x) + c_M$ [53; 24; 9].

We now show that $C(x)$ cannot be significantly greater than $|x|$.

Lemma 5.1.2. *There is a constant $c_0 \in \mathbb{N}$ such that for all $x \in \{0,1\}^*$,*

$$C(x) \leq |x| + c_0.$$

Proof. Let M be a Turing machine such that $M(x) = x$ for all $x \in \{0,1\}^*$, and let $c_0 = c_M$ be the optimality constant for M given by the optimality of our universal Turing machine. Then for all $x \in \{0,1\}^*$,

$$\begin{aligned} C(x) &\leq C_M(x) + c_M \\ &= |x| + c_M \\ &= |x| + c_0. \end{aligned}$$

□

An important concept is that of incompressibility or that there are strings that cannot be described with a simple Turing machine.

Definition 5.1.3. *For each constant c we say that a string x is* c-incompressible *if $C(x) \geq \log(x) - c$.*

The theorem most relevant to our purpose is the Incompressibility Theorem [24].

Theorem 5.1.4. *Let c be a positive integer. Every finite set A of cardinality m has at least $m(1 - 2^{-c}) + 1$ elements x with $C(x) \geq \log m - c$.*

Proof. The number of programs of length less than $\log m - c$ is

$$\sum_{i=0}^{\log m - c - 1} 2^i = 2^{\log m - c} - 1.$$

Hence, there are at least $m - m2^{-c} + 1$ elements in A that have no program of length less than $\log m - c$. □

Informally, the Incompressibility Theorem states that, given any finite set, there always exists high complexity elements.

The following theorem is useful in applying the Incompressibility Theorem to regular languages. It improves existing proofs by strengthening a bound of $2^{c+O(1)}$ [10] to 2^{c+1} while using a simpler argument.

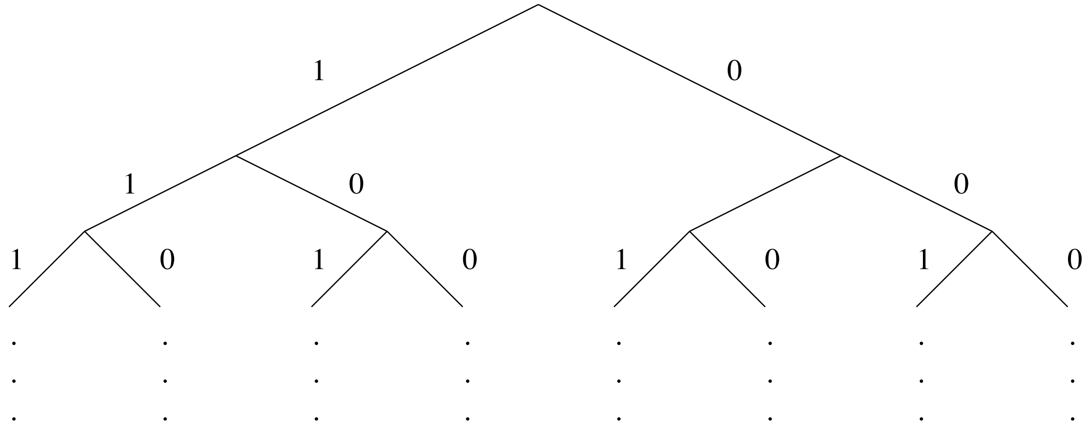


Figure 5.1 Representation of sequences as paths in a binary tree.

Theorem 5.1.5. *Let $c \in \mathbb{N}$.*

1. *There are infinitely many positive integers n for which at most 2^{c+1} strings $x \in \{0,1\}^n$ satisfy $C(x) \leq c + \log n$.*

2. *There are at most 2^{c+1} sequences $S \in \{0,1\}^\infty$ such that, for all $n \in \mathbb{N}$, $C(S[0 \ldots n-1]) \leq c + \log n$.*

Proof. Let $c \in \mathbb{N}$.

1. For each $n \in \mathbb{N}$, let

$$B_n = \{x \in \{0,1\}^n \mid C(x) \leq c + \log n\}.$$

For each $m \in \mathbb{Z}^+$, define the average

$$b_m = \frac{1}{m} \sum_{n=1}^{m} |B_n|$$

of the cardinalities $|B_1|, \ldots, |B_m|$. Then, for every m we have

$$\begin{aligned}
b_m &= \frac{1}{m} \sum_{n=1}^{m} |B_n| \\
&= \frac{1}{m} \left| \bigcup_{n=1}^{m} B_n \right| \\
&\leq \frac{1}{m} \left| \{0,1\}^{\leq c + \log m} \right| \\
&< \frac{1}{m} 2^{c + \log m + 1} \\
&= 2^{c+1} \left(\frac{2^{\log m}}{m} \right) \\
&= 2^{c+1}
\end{aligned}$$

In order for this to hold for *every* m, there must be infinitely many n for which $|B_n| \leq 2^{c+1}$. For example, if there are a number of sets whose cardinality is larger than

2^{c+1}, it must be the case that there is a number of other sets of cardinality less than 2^{c+1} to ensure that the average cardinality at most 2^{c+1}.

2. Figure 5.1 is a representation of sequences in $S \in \{0,1\}^\infty$ where each sequence is represented by an infinite path in a binary tree where each choice is whether the next bit of the sequence is 0 or 1. Strings are represented by finite paths in this tree and each level of the tree includes all strings of a given length.

Now note that every subsequence $S[0 \ldots n-1]$ is a string of length n. So, according to the above, there are infinitely n for which 2^{c+1} strings $x \in \{0,1\}^n$ satisfy $C(x) \leq c + \log n$. This means that, when we require our sequences S to be such that for all $n \in \mathbb{N}$, $C(S[0 \ldots n-1]) \leq c + \log n$, there are infinitely many n where there are at most 2^{c+1} sequences in S. In other words, while there are an infinite number of sequences, part 1 tells us that only 2^{c+1} sequences can obey the constraint $C(S[0 \ldots n-1]) \leq c + \log n$ for all n (particularly including subsequence lengths n where there are only 2^{c+1} subsequences following this constraint).

□

It is important for many applications of our main topic, the KC Regularity Lemma, to know that there are high-complexity strings no matter the length of the string. The following Lemma and Corollary establish this.

Lemma 5.1.6. *For any $n, r \in \mathbb{N}$, if we choose a string $x \in \{0,1\}^n$ according to the uniform probability measure, then*

$$\text{Prob}[C(x) < |x| - r] < 2^{-r}.$$

Proof.

$$\begin{aligned}
\mathrm{Prob}[C(x) < |x| - r] &= 2^{-n}|\{x \in \{0,1\}^n \mid C(x) < n - r\}| \\
&\leq 2^{-n}|\{\pi \in \{0,1\}^* \mid |\pi| < n - r\}| \\
&= 2^{-n}|\{0,1\}^{n-r}| \\
&= 2^{-n}(2^{n-r} - 1) \\
&< 2^{-r}.
\end{aligned}$$

□

Corollary 5.1.7. *For every $n \in \mathbb{N}$, there exists $x \in \{0,1\}^n$ such that $C(x) \geq |x|$.*

Proof. Taking $r = 0$ in Lemma 5.1.6 gives $\mathrm{Prob}[C(x) < |x|] < 1$. □

We now show that Kolmogorov complexity increases without bound.

Lemma 5.1.8. $\lim_{n\to\infty} C(s_n) = \infty$.

Proof. Let $m \in \mathbb{N}$, and let

$$A_m = \{x \in \{0,1\}^* \mid C(x) \leq m\}.$$

Then

$$A_m = \{U(\pi) \mid \pi \in \{0,1\}^{\leq m} \text{ and } U(\pi) \text{ halts}\}$$

by definition of $C(x)$ so A_m is finite. This means there exists $n_m \in \mathbb{N}$ such that $A_m \subseteq \{s_0, s_1, \ldots, s_{n_m-1}\}$. Then, for all $n \geq n_m$, $s_n \notin A_m$ so $C(s_n) > m$. This implies that $C(s_n)$ grows without bound. □

In addition to increasing without bound, the function $f(n) = C(s_n)$ is "almost continuous" as a function of n. This is illustrated by the following theorem that shows that no computable function can add more than a bounded amount of information to its inputs and the corollary based on it (when that function is the successor function).

Theorem 5.1.9. *If $g : \{0,1\}^* \dashrightarrow \{0,1\}^*$ is computable, then there is a constant $c_g \in \mathbb{N}$ such that, for all $x \in dom(g)$,*

$$C(g(x)) \leq C(x) + c_g.$$

Proof. Let g be as given. Let M be a Turing machine such that

$$M(\pi) = g(U(\pi))$$

for all $\pi \in \{0,1\}^*$, and let $c_g = c_M$ where c_M is the optimality constant for M. We know that such a machine M exists because g is computable and, by definition, all operations of U are computable so, for example, M could be just running U on π then applying function g to the result.

To complete the proof, let $x \in dom(g)$. Fix a program $\pi \in \{0,1\}^*$ such that $U(\pi) = x$ and $|\pi| = C(x)$. That is, π is the shortest program for U that produces x. Then

$$M(\pi) = g(U(\pi)) = g(x),$$

so

$$C_M(g(x)) \leq |\pi| = C(x)$$

by selection of π and finally

$$\begin{aligned} C(g(x)) &\leq C_M(g(x)) + c_M \\ &\leq C(x) + c_M \\ &= C(x) + c_g. \end{aligned}$$

□

Corollary 5.1.10. *There is a constant $c \in \mathbb{N}$ such that for all $n \in \mathbb{N}$, $|C(s_n) - C(s_{n+1})| \leq c$.*

Proof. Apply Theorem 5.1.9 to the function $g(s_n) = s_{n+1}$ and let $c = c_g$. □

5.2 The Regularity Theorem

We now begin our discussion of how Kolmogorov complexity can be applied to show a language is not regular. In this discussion, the instructor can either introduce the Myhill-Nerode theorem separately or simply include the theorem and it's proof as part of the proof of the KC Regularity Lemma. We include both approaches.

We will assume the following format and terminology for the Myhill-Nerode theorem when it is included previously in the course [19; 1].

Theorem 5.2.1. *A language $L \subseteq \{0,1\}^*$ is regular iff it is the union of equivalence classes of a right-invariant equivalence relation of finite index on Σ^*.*

Informally, we can view this theorem in the context of DFAs where each equivalence class is represented by a state of the DFA and the final state is the equivalence class of strings in L. We will denote the equivalence class of $x \in \Sigma^*$ under a relation (which is made clear from context) as $[x]$. The requirement of right-invariance means that, if two strings x and y are in the same equivalence class and we add the same string w to each, then xw and yw are also in the same equivalence class. In the DFA analogy, x and y are in the same DFA state so transitioning according to the same string w will result in the same new state.

To illustrate the approach of the KC Regularity Lemma, we start with one of the simplest languages used to show nonregularity in undergraduate computational theory courses: $L = \{0^n 1^n \mid n \geq 1\}$.

To see this language is nonregular by directly application of the Myhill-Nerode Theorem, we must show that there is an infinite number of ways to index the right-invariant equivalence relation over L. Pick any string 0^i where $i \geq 0$. The right-invariant equivalence class $[0^i]$ includes all strings y such that, for all $z \in \Sigma^*$, $0^i z \in L$ iff $yz \in L$. When n is set to a particular value, there is only one z for each i such that $0^iz \in L$ so each equivalence class is a singleton. However, L includes all values of n and there is one member of this equivalence class per value of n so new equivalence classes exist for each value of n. This means the index of the right-invariant equivalence relation over L is infinite.

This method of directly applying the theorem is generally difficult to convey to students as it conveys little of the intuition of how regular languages work. Let us examine L using the tools of Kolmogorov complexity.

Assume L is regular and D is a witness DFA to this fact. Consider processing any 0^n1^n (for set value of n). Then there is a state q in D that we are in after n steps of an acceptable string (i.e. after processing 0^n). Then q and D form a description of n: by running D initialized to state q on input consisting only of 1s, the first time D enters an accepting state is after n consecutive $1s$ [29]. By our assumption, D accepts exactly L and the size of the description of D and q is bounded by a constant c that is not dependent on n. So we can compute n using a Turing machine that takes as input this combination of D and q which means that $C(n) \leq c + O(1)$. But, by choosing n with $C(n) \geq \log(n)$ via Theorem 5.1.4, we obtain a contradiction for all large enough n. This means that no such DFA D can exist and L is not regular. This approach mirrors more closely the human approach of trying to build a finite state machine that always requires more states as n increases to compute the language.

To formally generalize this application of Kolmogorov complexity, we show the equivalence of regular languages and languages whose characteristic sequence is of low complexity. We now define some terms used in this process.

Given a language A and a string $x \in \Sigma^*$, define $A_x = \{y \in \Sigma^* \mid xy \in A\}$. We are interested in two representations of the languages A_x: its characteristic sequence χ_{A_x} and list representation $\mathcal{L}_{A_x}$. We now define the characteristic sequence and list representation of any language A.

Definition 5.2.2. *The* characteristic sequence *of a language $A \subseteq \Sigma^*$ is the sequence $\chi_A \in \{0,1\}^\infty$ whose i^{th} bit is*

$$\chi_A[i] = [\![w_i^\Sigma \in A]\!]$$

for all $i \in \mathbb{N}$.

Informally, a characteristic sequence is a series of 0s and 1s where each 1 indicates a member of A.

Definition 5.2.3. *Let $w_{n_0}^\Sigma, w_{n_1}^\Sigma, \ldots$ be the enumeration of A in standard order (i.e., $n_0 < n_1 < n_2 < \ldots$). Then, for each $i \in \mathbb{N}$,*

$$x_i = \begin{cases} 0w_{n_i}^\Sigma & \text{if } 0 \le i < |A| \\ \lambda & \text{if } i \ge |A|. \end{cases}$$

The list representation *of a language $A \subseteq \Sigma^*$ is the infinite sequence*

$$\mathcal{L}_A = (x_0, x_1, x_2, \ldots)$$

of strings x_i.

For each $n \in \mathbb{N}$, we also define the string $\mathcal{L}_A[n] \in \{0,1\}^*$ by the recursion

$$\begin{aligned} \mathcal{L}_A[0] &= \lambda \\ \mathcal{L}_A[n+1] &= \mathcal{L}_A[n]0^{|x_n|}1x_n. \end{aligned}$$

Call $\mathcal{L}_A[n]$ the binary encoding of the list *of the first n items in the list $\mathcal{L}_A$.*

We are now equipped to characterize regular languages using Kolmogorov complexity. In the following result due to Li and Vitányi [29], we write $C(n)$ as $C(s_n)$ and vice versa.

Theorem 5.2.4. *For each language $A \subseteq \Sigma^*$, the following conditions are equivalent.*

(1) *A is regular.*

(2) *There is a constant $a_A \in \mathbb{N}$ such that, for all $x \in \Sigma^*$ and $n \in \mathbb{N}$,*

$$C(\mathcal{L}_{A_x}[n]) \leq a_A + C(n).$$

(3) *There is a constant $b_A \in \mathbb{N}$ such that, for all $x \in \Sigma^*$ and $n \in \mathbb{N}$,*

$$C(\chi_{A_x}[0 \ldots n-1]) \leq b_A + C(n).$$

(4) *There is a constant $c_A \in \mathbb{N}$ such that, for all $x \in \Sigma^*$ and $n \in \mathbb{N}$,*

$$C(\chi_{A_x}[0 \ldots n-1]) \leq c_A + \log n.$$

Proof. (**1**) ⇒ (**2**). Let A be regular. Then there is a DFA $M = (Q, \Sigma, \delta, s, F)$ such that $L(M) = A$. Without loss of generality, assume that $Q \subseteq \{0,1\}^m$ for some $m \in \mathbb{N}$.

There is a straightforward algorithm that, given $q \in Q$, determines whether there is a path from q to a state in F that crosses a loop. Specifically, start in state q then simulate the transition for each possible input. Track what states this exploration visits (and has visited in the past) and try all possible transitions from each successor state. The key is that, once we revisit a state, do not try all possible transitions from that state. Eventually we will visit all states reachable from q as well as discovering states on a loop from q.

This algorithm thus determines whether the set

$$A(q) = \{y \in \Sigma^* \mid \widehat{\delta}(q, y) \in F\}$$

is infinite. It is straightforward to construct a Turing machine $\widehat{M}$ that, on input $q\pi$, where $q \in Q, \pi \in \{0,1\}^*$, and $U(\pi) = s_n$, outputs the string $\mathcal{L}_{A(q)}[n]$. Let $a_A = m + \widehat{c}$, where $\widehat{c}$ is an optimality constant for $\widehat{M}$ and m is the log of the number of states in $\widehat{M}$.

To see that **(2)** holds, let $x \in \Sigma^*$ and $n \in \mathbb{N}$. Let $q = \widehat{\delta}(s, x)$, and let $\pi \in \{0,1\}^*$ be such that $U(\pi) = s_n$ and $|\pi| = C(n)$. Then

$$\widehat{M}(q\pi) = \mathcal{L}_{A(q)}[n] = \mathcal{L}_{A_x}[n],$$

so

$$\begin{aligned} C(\mathcal{L}_{A_x}[n]) &\leq C_{\widehat{M}}(\mathcal{L}_{A_x}[n]) + \widehat{c} \\ &\leq |q\pi| + \widehat{c} \\ &= m + C(n) + \widehat{c} \\ &= a_A + C(n), \end{aligned}$$

i.e., **(2)** holds.

(**2**) ⇒ (**3**). Assume that **(2)** holds, with a_A as witness. Define a partial function $g : \{0,1\}^* \dashrightarrow \{0,1\}^*$ as follows. Let a string $u \in \{0,1\}^*$ be in the domain of g if it has the form

$$u = 0^{|y_0|}1y_0 0^{|y_1|}1y_1 \ldots 0^{|y_k|}1y_k$$

for some $k \in \mathbb{N}$, and there exists $0 \leq j \leq k$ such that

(i) for each $0 \leq i < j$, y_i begins with a 0;

(ii) $y_0, \ldots, y_{j-1}$ appear in standard order; and

(iii) for each $j \leq i < k$, $y_i = \lambda$.

If u is in the domain, then $g(u)$ is the unique string $v \in \{0,1\}^k$ with the following properties.

(a) For each $0 \leq i < j$, if $y_i = 0s_{n_i}$, where $n_i < k$, then $v[n_i] = 1$.

(b) All other bits of v are 0.

Informally, this function g translates a list representation to a characteristic sequence. g is clearly computable so Theorem 5.1.9 tells us that there is a constant $c_g \in \mathbb{N}$ such that, for all $u \in dom(g)$,

$$C(g(u)) \leq C(u) + c_g.$$

Let $b_A = a_A + c_g$. To see that **(3)** holds, let $x \in \Sigma^*$ and $n \in \mathbb{N}$. Then

$$g(\mathcal{L}_{A_x}[n]) = \chi_{A_x}[0 \ldots n-1],$$

so

$$\begin{aligned} C(\chi_{A_x}) &= C(g(\mathcal{L}_{A_x}[n])) \\ &\leq C(\mathcal{L}_{A_x}[n]) + c_g \\ &= a_A + C(n) + c_g \\ &= b_A + C(n), \end{aligned}$$

i.e., **(3)** holds.

(**3**) ⇒ (**4**). Assume that **(3)** holds. Let $c_A = b_A + c_0 + 1$, where c_0 is the constant from Lemma 5.1.2 that limits the complexity of x. Then, for all $x \in \Sigma^*$ and $n \in \mathbb{Z}^+$,

$$\begin{aligned} C(\chi_{A_x}[0 \ldots n-1]) &\leq b_A + C(s_n) \\ &\leq b_A + |s_n| + c_0 \\ &= b_A + \lfloor \log(n+1) \rfloor + c_0 \\ &\leq b_A + c_0 + 1 + \log n \\ &= c_A + \log n, \end{aligned}$$

i.e., **(4)** holds. The second equality is due to Lemma 5.1.2. A similar proof can be used to show that there is a constant $a'_A \in \mathbb{N}$ such that, for all $x \in \Sigma^*$ and $n \in \mathbb{N}$, $C(\mathcal{L}_{A_x}[n]) \leq a'_A + \log n$.

(**4**) ⇒ (**1**). Assume that **(4)** holds. Note that χ_{A_x} is an infinite sequence and we require that prefix strings of this sequence follow the same inequality as specified in Theorem 5.1.5 (part 2) with $c = c_A$. Since there is a finite number of such sequences χ_{A_x}, the set

$$\phi_A = \{A_x \mid x \in \Sigma^*\}$$

is finite (specifically, it has 2^{c+1} members). Define a relation $\approx$ on Σ^* by

$$x \approx x' \Leftrightarrow A_x = A_{x'}.$$

Then $\approx$ is an equivalence relation and has finite index, i.e., only finitely many equivalence classes as ϕ_A is finite. Also, for all $x, x', y \in \Sigma^*$,

$$\begin{aligned} x \approx x' &\Rightarrow A_x = A_{x'} \\ &\Rightarrow (\forall w) xw \in A \Leftrightarrow x'w \in A \\ &\Rightarrow (\forall z) xyz \in A \Leftrightarrow x'yz \in A \\ &\Rightarrow A_{xy} = A_{x'y} \\ &\Rightarrow xy \approx x'y, \end{aligned}$$

i.e., $\approx$ is right-invariant. If the Myhill-Nerode theorem is discussed in the class previously, this is enough to show that A is regular. However, in case the Myhill-Nerode theorem is not discussed, we now show A is regular directly.

For each $x \in \Sigma^*$, let $[x]$ be the $\approx$-equivalence class of x. Let $M = (Q, \Sigma, \delta, s, F)$ where

$$\begin{aligned} Q &= \{[x] \mid x \in \Sigma^*\}, \\ s &= [\lambda], \\ F &= \{[x] \mid x \in A\}, \end{aligned}$$

and $\delta : Q \times \Sigma \to Q$ is defined by

$$\delta([x], a) = [xa]$$

for all $a \in \Sigma$. Then Q is finite (because $\approx$ has finite index) and δ is well-defined (because $\approx$ is right-invariant), so M is a DFA. An easy induction shows that, for all $[x] \in Q$ and $y \in \Sigma^*$,

$$\widehat{\delta}([x], y) = [xy].$$

It follows that, for all $x \in \Sigma^*$,

$$\begin{aligned} x \in L(M) &\Leftrightarrow \widehat{\delta}(s, x) \in F \\ &\Leftrightarrow \widehat{\delta}([\lambda], x) \in F \\ &\Leftrightarrow [\lambda x] \in F \\ &\Leftrightarrow [x] \in F \\ &\Leftrightarrow x \in A, \end{aligned}$$

whence $L(M) = A$. Hence A is regular. □

From now on, we write $C(w_n^\Sigma) = C(s_n)$. The following corollary of this result we call the Kolmogorov complexity (KC) regularity lemma [29].

Corollary 5.2.5. *If $A \subseteq \Sigma^*$ is regular, then there is a constant $d_A \in \mathbb{N}$ such that, for all $x, y_n^x \in \Sigma^*$, if y_n^x is the n^{th} string in A_x (counting from 0 in the standard ordering of Σ^*), then*

$$C(y_n^x) \leq d_A + C(n).$$

Proof. For any language $B \subseteq \Sigma^*$, define partial function

$$h_B : \{0,1\}^* \dashrightarrow \{0,1\}^*$$

such that, if $u = \mathcal{L}_B[n+1]$ and $|B| > n$, then $h_B(u) = w_n^\Sigma$, where w_n^Σ is the n^{th} element of B. (If u is not of this form, then $h(u)$ is undefined.) Then h_B is computable, so there is a constant c_h for h_B as in Theorem 5.1.9. Informally, h_B simply extracts the last element of B from a list representation of B.

Assume that A is regular. Then there is a constant $a_A \in \mathbb{N}$ as in condition (2) of Theorem 5.2.4. Let $d_A = a_A + c_h + c$ with c as the gap between $C(s_n)$ and $C(s_{n+1})$ identified in Corollary 5.1.10. Then, for all $x, y_n^x \in \Sigma^*$, if y_n^x is the n^{th} string in A_x, we have

$$\begin{aligned} C(y_n^x) &= C(h_A(\mathcal{L}_{A_x}[n+1])) \\ &\leq C(\mathcal{L}_{A_x}[n+1]) + c_h \\ &\leq a_A + C(n+1) + c_h \\ &\leq a_A + C(n) + c + c_h \\ &= d_A + C(n). \end{aligned}$$

□

The intuition behind the Kolmogorov complexity (KC) regularity lemma is that a regular language A can be computed by a DFA M and, to compute a y_n^x, we start in the state $\widehat{\delta}(q_0, x)$ (where q_0 is the initial state of M) and count off n steps to find y_1^x, y_2^x, . . . , y_n^x.

5.3 Usage Examples

In order to most conveniently apply the results of the previous section, we restate Corollary 5.2.5 in the following contrapositive form.

Corollary 5.3.1. *If a language $A \subseteq \Sigma^*$ has the following property, then it is* not *regular.*

For all $d \in \mathbb{N}$, there exists $x, y_n^x \in \Sigma^*$ and $n \in \mathbb{N}$ such that y_n^x is the n^{th} string in A_x and $C(y_n^x) > d + C(n)$.

In other words, this property states that there is an $xy_n^x \in A$ that cannot be computed simply by running a DFA — for some reason that differs in each proof using this Corollary, we need a longer Turing machine program than a regular language does. We now apply this corollary to several elementary examples of nonregular languages.

Example 5.3.2. $A = \{0^n 1^n \mid n \in \mathbb{N}\}$ is not regular.

Proof. Let $d \in \mathbb{N}$. By Lemma 5.1.8, there exists $k \in \mathbb{N}$ such that $C(1^k) > d + C(0)$ (where $s_n = 1^k$ in the lemma). By applying Corollary 5.3.1 with $x = 0^k$, $y = 1^k$, and $n = 0$, A is not regular. □

Example 5.3.3. $B = \{0^p \mid p \in \mathbb{N} \text{ is prime}\}$ is not regular.

Proof. Let $d \in \mathbb{N}$. It is well known that there are arbitrarily large gaps in the primes. This is because, for any $m \in \mathbb{Z}^+$, all n with $m!+2 \leq n \leq m!+m$ are composite. By Lemma 5.1.8, there exist consecutive primes p, q such that $C(0^{q-p}) > d+C(1)$. By applying Corollary 5.3.1 with $x = 0^p$, $y = 1^{q-p}$, and $n = 1$, A is not regular. □

Example 5.3.4. $B = \{0^k 1^\ell \mid \gcd(k, \ell) = 1\}$ is not regular.

Proof. Let $d \in \mathbb{N}$. By Lemma 5.1.8, there exists a prime p such that $C(1^{p-1}) > d + C(1)$. By applying Corollary 5.3.1 with $x = 0^{(p-1)!}$, $y = 1^{p-1}$, and $n = 1$, A is not regular. □

5.4 Comparison with Pumping Lemmas

We now discuss some of the history of the pumping lemma, an alternate method of showing a language is nonregular, to see why the KC Regularity lemma (Corollary 5.2.5) is more intuitive and/or useful. First, we define what a *pump* is.

Definition 5.4.1. *Let $L \in \Sigma^*$, $x \in \Sigma^*$, and $x = uvw$, then v is a* pump *for x relative to L iff, for all $i \geq 0$, $u(v)^i w \in L$ iff $x \in L$.*

The following is often referred to as the Pumping Lemma.

Lemma 5.4.2. *Let $A \subseteq \Sigma^*$ be a regular set. Then the following property holds of A.*

There exists $k \geq 0$ such that for any strings $xyz \in A$ and $|y| \geq k$, there exist strings u, v, w such that $y = uvw$, $v \neq \lambda$, and for all $i \geq 0$, the string xuv^iwz is in A (i.e., v is a pump for yz relative to L).

It can be easily proven that this is a property of all regular languages [25]. This easy explanation is key to developing any method of showing nonregularity for an undergraduate course. It is most often used to show nonregularity through its contrapositive form.

While this pumping property is necessary for all regular languages, it is not sufficient. We can show that this pumping condition does not even imply a language is recursive.

Theorem 5.4.3 ([12])**.** *There are $2^{\aleph_0}$ languages which satisfy the pumping condition (Lemma 5.4.2).*

Proof. Let $\Sigma = \{a_{i,j} \mid 0 \leq i, j \leq 3\}$. We define two maps $f_a, f_b : \Sigma \to \Sigma$ where

$$\begin{aligned} f_a(a_{i,j}) &= a_{i+1,j} \mod 4, \\ f_b(a_{i,j}) &= a_{i,j+1} \mod 4. \end{aligned}$$

The functions f_a, f_b are permutations of Σ and have the property that applying two functions can never have the same effect as applying one. This is because applying two functions can increment both subscripts i, j by one (mod 4) or one subscript by two (mod 4) and a single application of a function can never achieve this.

Let $n_1, n_2, \ldots, n_m, m \in \mathbb{N}$ and $\sigma_1, \sigma_2, \ldots, \sigma_m \in \Sigma$ defined recursively: $\sigma_1 = a_{0,0}$ and, for all $\ell < m$, $\sigma_{\ell+1}$ is either $f_a(\sigma_\ell)$ or $f_b(\sigma_\ell)$. Then define a string as "legal" if it is of the form $x = (\sigma_1)^{n_1}(\sigma_2)^{n_2} \ldots (\sigma_m)^{n_m}$.

Now let $\Sigma_1 = \{a, b\}$ and language $X \subseteq \Sigma_1^*$. If we consider each σ_ℓ as being the result of a "transition" due to f_a or f_b, there are $m - 1$ transitions and they correspond to a string y in Σ_1^*. So we say x encodes y. As an example, $x = a_{0,0}a_{1,0}a_{1,0}a_{1,1}$ is legal with $n_1 = 1$, $n_2 = 2$, $n_3 = 1$, and $y = ab$.

Let the "parity" of a string in Σ be the sum of all subscripts i, j (mod 2). Now we let $L(X) = \{x \mid x \text{ is legal and } x \text{ codes a } y \text{ such that } y \in X\} \bigcup \{x \mid x \text{ is illegal and the parity of } x \text{ is } 0\}$.

Clearly the map L is one-to-one. We shall now show that $L(X)$ always satisfies the pumping condition.

Let $k = 6$ and $zyz' \in \Sigma^{*}$[*] such that $|y| \geq 6$. We now consider three cases:

(1) zyz' is legal and y contains a double σ. Let $y = u\sigma w$ where the last symbol of u is also σ and let $v = \sigma$. Then, for all i, $zu(\sigma)^{i}wz'$ is legal and codes the same x that zyz' does. So the pumping condition holds.

(2) zyz' is legal but contains no double σ. We now have to examine parities in subcases. Say that $zyz' \in L(X)$ and has odd parity. Then y must itself contain a symbol of odd parity. Let σ be that symbol and $y = uvw$. We can choose v so that it is not the last symbol of y.

Then $i \geq 1$, $zu(v)^{i}wz'$ codes the same string x as zyz' and is legal so $zu(v)^{i}wz' \in L(X)$. For $i = 0$, $zu(v)^{i}wz' = zuwz'$ has zero parity and is illegal so again $zu(v)^{0}wz' \in L(X)$.

The remaining subcases where zyz' has even parity and/or $zyz' \notin L(X)$ are similar.

(3) zyz' is illegal. The illegality may be caused by the initial symbol being something other than $a_{0,0}$ or by a bad transition. In any case zyz' contains a subpiece y' of length ≤ 2 such that saving that piece will preserve illegality. Hence $|y| \geq 6$, we can find a v' such that (a) v' is disjoint from y' and (b) $|v'| = 2$.

Now let v be a substring of v' of parity 0. There must be a nontrivial such substring with one or two symbols. Let $y = uvw$. Then, for all $i \geq 0$, $zu(v)^{i}wz'$ has the same parity as zyz' and is illegal because $zu(v)^{i}wz'$ is in $L(X)$ iff zyz' is.

□

We can use this fact to identify a context-free, nonregular language that satisfies Lemma 5.4.2 by applying the transformation L given in the proof of Theorem 5.4.3 above to $\{a^{n}b^{n} \mid n \geq 0\}$.

Lemma 5.4.4 ([12]). *There exists a language X that is context-free and satisfies Lemma 5.4.2 but is not regular.*

[*] We use z and z' instead of the x and z used in Lemma 5.4.2 to avoid conflation with $x \in \Sigma^{*}$ mentioned earlier.

Proof. Consider $X = \{a^nb^n \mid n \geq 0\}$ and the strings y encoding them using the procedure given in the proof of Theorem 5.4.3 above. Specifically, the following context-free rules accept only $X_0 = \{a^nb^n \mid n \geq 0, n = 0 \mod 4\}$.

$$\begin{aligned}
S &\rightarrow A_{0,0}A_{1,0}A_{2,0}A_{3,0}SA_{0,1}A_{0,2}A_{0,3}A_{0,0} \\
S &\rightarrow \lambda \\
A_{i,j} &\rightarrow a_{i,j}A_{i,j} \\
A_{i,j} &\rightarrow \lambda
\end{aligned}$$

We can similarly define X_1, X_2, and X_3 and, as the finite union of context-free languages is context-free, X is context-free. By reasoning given in Theorem 5.4.3, $L(X)$ is also context free and satisfies the pumping condition given in Lemma 5.4.2.

But the set cannot be regular. Consider strings y_i such that y_i represents a^i, z_i represents b^i, and i is divisible by 4. Now, for all $i, j \in [0, 4] \cap \mathbb{N}$, if $i \neq j$ then $y_iz_i \in L(X)$ and $y_jz_i \notin L(X)$. Hence, by the Myhill-Nerode Theorem, $L(X)$ is not regular. □

There are more complicated-to-prove versions that are both necessary and sufficient conditions for regularity [20; 12; 56; 60]. Here is a first necessary and sufficient pumping condition [20] that is easy to prove but is difficult to apply.

Theorem 5.4.5. *L is regular if and only if there is a k such that, for all $x \in \Sigma^*$, if $|x| \geq k$, then $\forall u, v, w$ $x = uvw$, $v \neq \lambda$, and $\forall z$ v is a pump for xz relative to L. I.e., for all $i \geq 0$ and $z \in \Sigma^*$, $u(v)^iwz \in L$ iff $xz \in L$.*

Proof. The following proof emerges directly from the Myhill-Nerode Theorem [12].

It is sufficient to show that, if $\equiv$ is the Nerode equivalence relation over L then $\forall x$ $\exists x'$ such that $|x'| < k$ and $x \equiv x'$. Then as there are only finitely many x' with $|x'| < k$, $\equiv$ has finite index. It is sufficient to show this condition as follows: if $|x'| \geq k$ then there is an x'' such that $|x''| < |x|$ and $x \equiv x''$ as repeating this process will eventually yield the desired x'.

Now note that, given u, v, w, x as in the hypothesis, $x \equiv uw$ by setting $i = 0$ and $|uw| < |x|$.

□

We can show that this is true directly [20] as well. In both cases, the "only if" part of the theorem is true and follows easily. In Theorem 5.4.5, we must pump not just x but xz with $z \in \Sigma^*$ which makes it difficult to apply. We can also prove that the following, "local" *block pumping property* is necessary and sufficient for regularity as well [12].

Property 5.4.6. *$L \subseteq \Sigma^*$ has the* block pumping property *if there is a k such that, for all $x, w, y_1, \ldots, y_k, w'$ in Σ^*, if $x = wy_1 \ldots y_k w'$ then there exist l, j $1 \leq l \leq j \leq k$ such that $y_l y_{l+1} \ldots y_j$ is a pump for x relative to L.*

That this property is necessary and sufficient for regularity is shown by letting $i = 0$ in the property (called the *block cancellation property*) and applying a finite version of Ramsey's theorem from graph theory [12]. It was later shown that the language of a finitely generate free monoid is regular iff it satisfies the positive block pumping property, Property 5.4.6 with $i > 0$ [60]. However, the proof of both the block pumping property's and Theorem 5.4.5's equivalence to regular languages is involved and not easily taught at the undergraduate level.

CHAPTER 6. ESSENTIAL HIDDEN VARIABLES IN BAYESIAN NETWORKS

In this chapter, we change gears to examine how an aspect of a model can be viewed under a different light. Specifically, we investigate a new way to view hidden variables.

After an example to motivate our discussion in Section 6.1, we begin our examination of hidden variables in Bayesian networks with notation and definitions (Section 6.2) including new terminology to characterize hidden variables. We present an exploratory algorithm that systematically examines all networks of a size for the existence of essential hidden variables. Since the number of such networks is exponential in that size, we also present a number of optimizations (Section 6.3). We conclude with the experimental results of these algorithms and verification that the constraint-based approach [39] yields the same results as systematic approaches.

Through an examination of all networks up to size 8, we discover that a Bayesian network that has an essential hidden variable must have an embedded "W network" edge set (see Section 6.2 for definition). Although we are not able to produce any general independence-based characterization of the relationships that must hold for an essential variable to be present, our results for small networks can be used in future work to establish general necessary and sufficient conditions for essential hidden variables.

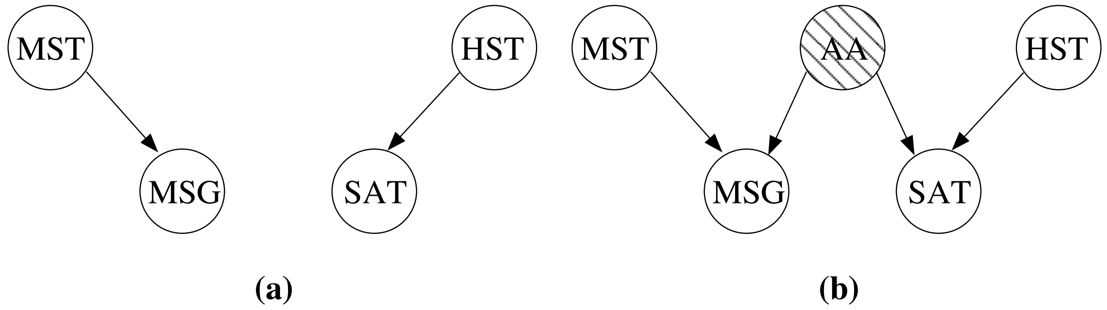


Figure 6.1 Two Bayesian networks representing the discussed data (a) without and (b) with hidden variable AA, shown shaded.

6.1 Motivating Example

Consider the case of modeling four attributes from data collected from high school graduates: middle school grades (MSG), average middle and high school teachers' rating (MST and HST, respectively) for the teachers the student had, and the student's SAT score (SAT)*. Let us say that a set of data collected gives the following situation: For a randomly selected student for whom no information is known about their MSG, their MST does not influence (has no statistical dependence with) their SAT score. However, for a randomly chosen student with a high MSG, they are more likely to do well on their SATs if MST is high. In addition, for a randomly chosen student with low MSG, they tend to do poorly on their SATs if their MST is low.

This situation may occur if a student that simply has good teachers in middle school (high MST) are no more or less likely to do well on the SATs than another other student with worse middle school teachers. However, the data may indicate that those students who have good grades in middle school (high MSG) in addition to good middle school teachers (high MST) will do well on the SATs compared to other students with low grades and good teachers or good grades but poor teachers.

A Bayesian network representing this situation is depicted in Figure 6.1(a). While this

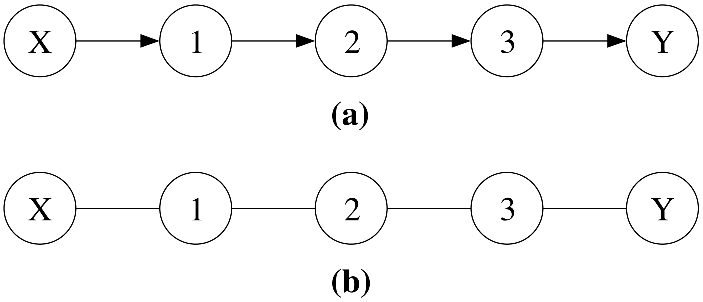
*The SAT, or Scholastic Aptitude Test$^{\mathrm{TM}}$, is a standard test administered by the Educational Testing Service (ETS$^{\mathrm{TM}}$) and used by colleges to decide whether to admit undergraduate students.

network correctly represents the hypothesis that the middle school teacher ratings (MST) are (unconditionally) unrelated to the SAT score, it does not have a mechanism for representing the relationship between MST and SAT given MSG. This relationship can be represented by adding a hidden variable we label AA ("academic aptitude") as shown in Figure 6.1(b). Even though AA cannot be measured, the existence of an attribute between middle school grades (MSG) and SAT score allows representation of the relationship between MST and SAT score given a high MSG. Intuitively, a high MSG and high MST indicates a high AA which, in turn, indicates that the student will do well on the SAT. Conversely, a low MSG and low MST indicates a low AA which tends to adversely affect the SAT score.

If a few more relationships hold in the data, we show in this chapter that AA is actually an essential hidden variable. This means that no network over the 4 measured attributes $\{MST, MSG, HST, SAT\}$ can represent all the relationships described by our hypothetical data without a hidden variable. Specifically, methods we examine give a way to verify that the structure implied by our data cannot be represented by a Bayesian network of size 4. The network in Figure 6.1(b) therefore is a more accurate representation of our data distribution so inference using this network will be more accurate than the network in Figure 6.1(a). We call this hidden variable essential to representing the distribution with a Bayesian network (i.e., an essential hidden variable) because the addition of a hidden variable enables representation of the underlying distribution more accurately than a smaller network can.

6.2 Bayesian Network Notation

This section describes terminology used in the discussion of essential hidden variables.



(a)

(b)

Figure 6.2 Paths from X to Y through simple path $\mathbf{P} = \{1{,}2{,}3\}$ in an (a) undirected and (b) directed graph.

6.2.1 Basic Graph Terminology

The translation of a distribution to a graph maps each attribute to a vertex in the graph. Edges in the graph represent relationships between attributes where the nature of the relationship depends on the graphical model. We will refer to attributes and vertices interchangeably in this chapter. Bold letters denote sets of attributes. Capital letters denote variable names (X, Y, Z, etc.), while small letters denote values for these variables (x, y, z, etc.).

In a directed graph, a *directed simple path* P between two distinct vertices X and Y refers to a list of unique vertices that lie between X and Y such that there is an edge from X to the first vertex, between the first vertex and the second, etc. up to having an edge from the last vertex in the list to Y (Figure 6.2(a)). A *simple path* in an undirected graph is the same except that the edges are undirected (Figure 6.2(b)). We may also refer to *undirected paths* in directed graphs — these are paths on the graph that result from ignoring the directions of edges. There cannot be any cycles in either type of simple path because repeat vertices are not allowed.

Given a vertex V in a directed graph, we can identify several classes of related vertices. All vertices with a directed edge to V are called *parents* of V and denoted as $Par(V)$, and all vertices with a directed edge from V are called *children* of V. The set of *descendants* of

V, denoted as $\mathit{Desc}(V)$, is defined recursively as follows: (a) V is in $\mathit{Desc}(V)$ and (b) any child of a vertex in $\mathit{Desc}(V)$ is in $\mathit{Desc}(V)$. In Figure 6.2(a), all vertices are descendants of X.

6.2.2 Independence Notation

Attribute sets $\boldsymbol{X}$ and $\boldsymbol{Y}$ are said to be *(conditionally) independent given attribute set* $\boldsymbol{Z}$ (denoted $\boldsymbol{X} \perp\!\!\!\perp \boldsymbol{Y} \mid \boldsymbol{Z}$) iff $\mathrm{P}_r(\boldsymbol{X} = \boldsymbol{x} \mid \boldsymbol{Y} = \boldsymbol{y}, \boldsymbol{Z} = \boldsymbol{z}) = \mathrm{P}_r(\boldsymbol{X} = \boldsymbol{x} \mid \boldsymbol{Z} = \boldsymbol{z})$ for all values $\boldsymbol{x}$ of $\boldsymbol{X}$, $\boldsymbol{y}$ of $\boldsymbol{Y}$ and $\boldsymbol{z}$ of $\boldsymbol{Z}$. This means that, in the subpopulation where $\boldsymbol{Z}$=$\boldsymbol{z}$, the value of $\boldsymbol{X}$ is not related to the value of $\boldsymbol{Y}$. $\boldsymbol{Z}$ is said to be *in evidence* with regard to this independence as we must have evidence that $\boldsymbol{Z}$=$\boldsymbol{z}$ for this independence between $\boldsymbol{X}$ and $\boldsymbol{Y}$ to hold.

$\boldsymbol{X}$ and $\boldsymbol{Y}$ are said to be *dependent given* $\boldsymbol{Z}$ (denoted $\boldsymbol{X} \not\perp\!\!\!\perp \boldsymbol{Y} \mid \boldsymbol{Z}$) if the conditions for $\boldsymbol{X} \perp\!\!\!\perp \boldsymbol{Y} \mid \boldsymbol{Z}$ do not hold. *Marginal independence* between $\boldsymbol{X}$ and $\boldsymbol{Y}$ is denoted by $\boldsymbol{X} \perp\!\!\!\perp \boldsymbol{Y}$ when the conditioning set is empty—similarly, *marginal dependence* is denoted $\boldsymbol{X} \not\perp\!\!\!\perp \boldsymbol{Y}$. We will abuse this notation slightly when a set is a singleton: instead of writing $\{X\} \perp\!\!\!\perp \{Y\} \mid \{Z\}$ we will write $X \perp\!\!\!\perp Y \mid Z$.

For attributes X, Y, and Z, $(X \perp\!\!\!\perp Y \mid Z)_P$ denotes that the independence relation $X \perp\!\!\!\perp Y \mid Z$ is reflected in the distribution P. For a graph G, $(X \perp\!\!\!\perp Y \mid Z)_G$ denotes $X \perp\!\!\!\perp Y \mid Z$ can be inferred by a well-defined set of rules such as d-separation (defined below).

6.2.3 Bayesian Network Formalisms

To define Bayesian networks and related properties, we utilize the Markov assumption.

Definition 6.2.1. *The* Markov Assumption *for a network* $G = (\boldsymbol{U}, \boldsymbol{E})$ *states that*

$$\forall X \in \boldsymbol{U},\ X \perp\!\!\!\perp [\boldsymbol{U} - \mathit{Desc}(X)] \mid \mathit{Par}(X).$$

Intuitively, the Markov Assumption states that each attribute is independent of its non-descendants given its parents. It is critical for many proofs involving Bayesian networks that the Markov Assumption hold.

The following formal definitions are adopted from [45].

Definition 6.2.2. *A graph G is an* independence map *(I-map) of distribution P over attributes $\boldsymbol{U}$ if there is a one-to-one correspondence between the elements of $\boldsymbol{U}$ and the vertices $\boldsymbol{U}$ of G such that for all disjoint subsets $\boldsymbol{X}$, $\boldsymbol{Y}$, $\boldsymbol{Z}$, $(\boldsymbol{X} \perp\!\!\!\perp \boldsymbol{Y} \mid \boldsymbol{Z})_G \Rightarrow (\boldsymbol{X} \perp\!\!\!\perp \boldsymbol{Y} \mid \boldsymbol{Z})_P$.*

Intuitively, a graph is a I-Map if all independences represented in the graph are represented in the distribution.

Definition 6.2.3. *A graph G is a* minimal I-map *of distribution P if no edges can be deleted from G without altering the property that G is an I-map of P.*

Definition 6.2.4. *Given a probability distribution P over a set of attributes $\boldsymbol{U}$, a directed acyclic graph (DAG) $D = (\boldsymbol{U}, \boldsymbol{E})$ is called a* Bayesian network *of P iff D is a minimal I-map of P.*

We will primarily be interested in distributions that are *faithful* [55]:

> If all and only the conditional independence relations true in [probability distribution] P are entailed by the Markov [assumption] applied to [graph] G, we will say that P and G are *faithful to one another.* We will, moreover, say that a distribution P is *faithful* provided there is some directed acyclic graph to which it is faithful. (page 13)

Many distributions are not faithful. As such, one goal of a procedure that generates a Bayesian network could be to get as close to a faithful Bayesian network for the input distribution as possible. Often this is balanced against a desire for simple networks, because these allow easier human understanding as well as contain conditional probability tables with fewer entries, leading to faster inference and less danger of overfitting the model to the data.

To derive the independences represented in Bayesian networks, we use the following *d-separation rules*:

$X \perp\!\!\!\perp 2 \mid \{1\}$	$X \perp\!\!\!\perp 2 \mid \{1,3\}$	$X \perp\!\!\!\perp 2 \mid \{1,Y\}$
$X \perp\!\!\!\perp 2 \mid \{1,3,Y\}$		
$X \perp\!\!\!\perp 3 \mid \{1\}$	$X \perp\!\!\!\perp 3 \mid \{2\}$	$X \perp\!\!\!\perp 3 \mid \{1,2\}$
$X \perp\!\!\!\perp 3 \mid \{1\}$	$X \perp\!\!\!\perp 3 \mid \{2\}$	$X \perp\!\!\!\perp 3 \mid \{1,2\}$
$X \perp\!\!\!\perp Y \mid \{1\}$	$X \perp\!\!\!\perp Y \mid \{1,2\}$	$X \perp\!\!\!\perp Y \mid \{1,2,3\}$
$X \perp\!\!\!\perp Y \mid \{2\}$	$X \perp\!\!\!\perp Y \mid \{2,3\}$	$X \perp\!\!\!\perp Y \mid \{3\}$
$X \perp\!\!\!\perp Y \mid \{1,3\}$		
$1 \perp\!\!\!\perp Y \mid \{2\}$	$1 \perp\!\!\!\perp Y \mid \{3\}$	$1 \perp\!\!\!\perp Y \mid \{2,3\}$
$1 \perp\!\!\!\perp Y \mid \{2,X\}$	$1 \perp\!\!\!\perp Y \mid \{3,X\}$	$1 \perp\!\!\!\perp Y \mid \{2,3,X\}$
$1 \perp\!\!\!\perp 3 \mid \{2\}$	$1 \perp\!\!\!\perp 3 \mid \{2,X\}$	$1 \perp\!\!\!\perp 3 \mid \{2,Y\}$
$1 \perp\!\!\!\perp 3 \mid \{2,X,Y\}$		
$2 \perp\!\!\!\perp Y \mid \{3\}$	$2 \perp\!\!\!\perp Y \mid \{3,1\}$	$2 \perp\!\!\!\perp Y \mid \{3,X\}$
$2 \perp\!\!\!\perp Y \mid \{3,1,X\}$		

Table 6.1 Independences present in the Bayesian network appearing in Figure 6.2(a).

Definition 6.2.5 ([45]). *Given a directed, acyclic graph G = ($\boldsymbol{U}$,$\boldsymbol{E}$), for all disjoint sets $\boldsymbol{X}$, $\boldsymbol{Y}$,$\boldsymbol{Z} \subseteq \boldsymbol{U}$, $(\boldsymbol{X} \perp\!\!\!\perp \boldsymbol{Y} \mid \boldsymbol{Z})_G$ if along every path between a node in $\boldsymbol{X}$ and a node in $\boldsymbol{Y}$ there is a node W satisfying one of the following two conditions: (1) W has converging arrows and none of $Desc(W)$ are in $\boldsymbol{Z}$, or (2) W does not have converging arrows and W is in $\boldsymbol{Z}$.*

An example of a Bayesian network for some distribution is given in Figure 6.2(a), where $\boldsymbol{U}$ $= \{X, Y, 1, 2, 3\}$ and $E = \{(X, 1), (1, 2), (2, 3), (3, Y)\}$. The Bayesian network of Figure 6.2(a) implies exactly the conditional independence relations given in Table 6.1.

6.2.4 Hidden Variables

Definition 6.2.6. *An attribute is said to be a* hidden variable *if nothing is known about the distribution of the attribute.*

Intuitively, hidden variables are an extreme form of missing data—hidden variables have all of their data missing. As such, hidden variables can never appear in any a statement about independence. However, note that the definition is more general—it includes situations where we have no knowledge of the parametric family of distributions that the hidden variable's

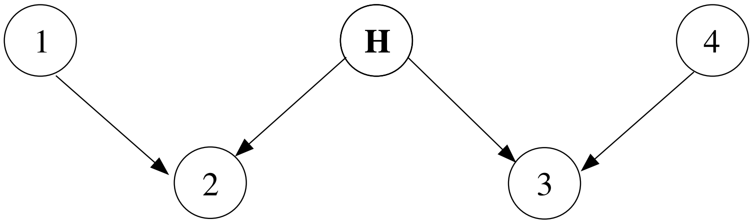


Figure 6.3 H labeled as a hidden variable in the W-network (attributes 1,2,3,4 are not hidden).

distribution is a member of, as well as when we also lack knowledge about the values of the parameters of the family. The concept of a hidden variable therefore includes attributes for which we do not even know the number of states of that variable.

Figure 6.3 depicts an example where H is a hidden variable. Independences generated from the graph with d-separation would include $1 \perp\!\!\!\perp 4 \mid 2$ but not ones about H and another variable ($H \perp\!\!\!\perp 4 \mid 2$) or with H in evidence ($2 \perp\!\!\!\perp 3 \mid H$).

There are two perspectives on the use of hidden variables in Bayesian networks. The first is to optimize the rank of a Bayesian network according to some score:

Definition 6.2.7. *An* optimizing hidden variable *in a Bayesian network $B = (U, E)$ given some scoring method S is a hidden variable H_O that, when added to B, provides a higher score $S(B')$, where $B' = (U \cup \{H_O\}, E')$, than any network of the same size as B without H_O.*

Optimizing hidden variables are usually introduced in the process of a search over the space of possible Bayesian networks that represent a set of independences. Most scoring methods for Bayesian networks explicitly balance the ability of the network to represent the data against the simplicity of the network.[†] By mandating a simple network for a data set (in terms of a low number of edges and hidden variables), we avoid overfitting our model to our data set and are more likely to create a model that is understandable by humans.

[†] Searching for the simplest possible network amongst networks that represent the input data well is often referred to as following *Occam's Razor*, that "one should not increase, beyond what is necessary, the number of entities required to explain anything"[54].

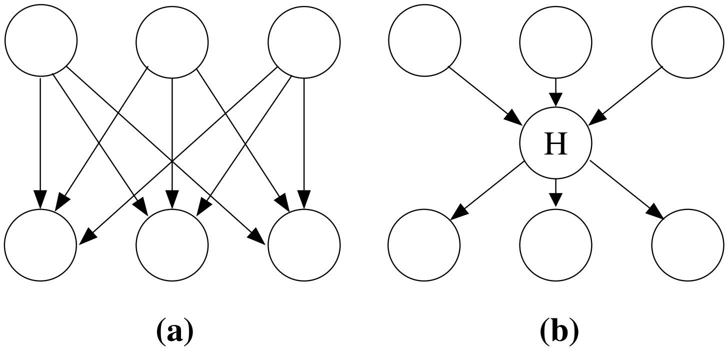


Figure 6.4 (a) Example network and (b) the same network with an optimizing hidden variable.

An example of an optimizing hidden variable is given in Figure 6.4. The three measured attributes located at the top of each graph are strongly related to the measured attributes at the bottom in both diagrams, but the network of Figure 6.4(b) is simpler. The optimizing hidden variable H reduces the size of the network by removing the need for many edges while not impacting the accuracy of the representation—the set of independences implied by the two Bayesian networks are identical. Most hidden-variable-aware scoring methods would choose Figure 6.4(b) over Figure 6.4(a).

Note that the class of distributions represented by both Bayesian networks in Figure 6.4 might be the same—an optimizing hidden variable was added to represent the same distribution more efficiently. This is not a necessary condition for an optimizing hidden variable—depending on the scoring method, the hidden variable may increase or decrease the number of represented independences.

In contrast, most algorithms in the social sciences focus on searching for what we call essential hidden variables.

Definition 6.2.8. *Consider any distribution P with n attributes $\boldsymbol{U}$ represented by a faithful Bayesian network $B(\boldsymbol{U'},\boldsymbol{E})$ where $\boldsymbol{U'} = \boldsymbol{U} \bigcup \{H_E\}$ for some hidden variable H_E. If, for the set of independences and dependences $\boldsymbol{S}_B$ implied by B, $\forall B' \in \{n$ attribute legal Bayesian networks representing P without hidden variables$\}$ and independences and dependences $\boldsymbol{S}_{B'}$*

implied by that network B', $\boldsymbol{S}_B \neq \boldsymbol{S}_{B'}$ and B is more faithful to P[‡] *than B', then H_E is an* essential hidden variable.

Informally, an essential hidden variable is a hidden variable H_E that, when added to the visible attributes of a Bayesian network, induces a set of properties that cannot exist in any network with the same number of visible attributes and no hidden variables. The primary question of this chapter is: Can a set of independences be exactly represented by a network with the same number of visible attributes and no hidden variable?

The essential quality of a hidden variable is derived from the fact that it does not merely simplify the network—it represents the distribution more accurately than any network structure that lacks it. The essential hidden variable enables us to more closely approach a faithful Bayesian network through correctly representing a larger number of independences from the distribution correctly.

In Figure 6.3, H is an example of an essential hidden variable—no Bayesian network with 4 attributes can represent the independence and dependence relationships between attributes 1 through 4. Establishing that this is the case is one of the results of our algorithm (explained in Section 6.3). We will call the structure of the network of Figure 6.3 the *W-network* and define its edge characteristics as follows.

Definition 6.2.9. *A* W-network *contains 4 measured attributes labeled as 1,2,3,4 and a hidden variable H. It satisfies the directed edge constraints* $(H, 2)$, $(H, 3)$, $(1, 2)$, $(4, 3)$, *no edge from H to 1 or from 1 to H, and no edge from H to 4 or from 4 to H.*

We can also discuss the W-network structure in the context of Verma's *P-Network* [55] using a non-independence constraint characterization [15].

[‡] *Although other criteria could be applied, we simply count the network representing more independences and dependences of P correctly (equally weighted).*

Definition 6.2.10. *The P-network is specified over visible attribute sets* $\boldsymbol{A}$*,* $\boldsymbol{B}$*,* $\boldsymbol{C}$*, and* $\boldsymbol{D}$[§]

$$\sum_{B}[\mathrm{P}_r(\boldsymbol{B} \mid \boldsymbol{A} = \boldsymbol{\mathit{0}})\mathrm{P}_r(\boldsymbol{D} \mid \boldsymbol{A} = \boldsymbol{\mathit{0}}, \boldsymbol{B}, \boldsymbol{C} = \boldsymbol{\mathit{0}}) - \mathrm{P}_r(\boldsymbol{B} \mid \boldsymbol{A} = \boldsymbol{\mathit{1}})\mathrm{P}_r(\boldsymbol{D} \mid \boldsymbol{A} = \boldsymbol{\mathit{1}}, \boldsymbol{B}, \boldsymbol{C} = \boldsymbol{\mathit{0}})] = 0$$

$$\sum_{B}[\mathrm{P}_r(\boldsymbol{D} \mid \boldsymbol{A} = \boldsymbol{\mathit{0}})\mathrm{P}_r(\boldsymbol{D} \mid \boldsymbol{A} = \boldsymbol{\mathit{0}}, \boldsymbol{B}, \boldsymbol{C} = \boldsymbol{\mathit{1}}) - \mathrm{P}_r(\boldsymbol{B} \mid \boldsymbol{A} = \boldsymbol{\mathit{1}})\mathrm{P}_r(\boldsymbol{D} \mid \boldsymbol{A} = \boldsymbol{\mathit{1}}, \boldsymbol{B}, \boldsymbol{C} = \boldsymbol{\mathit{1}})] = 0$$

If the sets $\boldsymbol{A}$, $\boldsymbol{B}$, $\boldsymbol{C}$, and $\boldsymbol{D}$ contain only a single variable each, the Verma constraints imply a 5-attribute network through a structure that is usually depicted as a W-network with an extra edge (thus the "P"). This definition relies on non-independence constraints and each variable having binary cardinality.

Note that essential hidden variables and optimizing hidden variables do not partition the space of possible hidden variables. Addition of an essential hidden variable may result in an increase in accuracy significant enough to justify adding the same hidden variable.

6.3 An Algorithm for Detecting Essential Hidden Variables

Given the theoretical significance of essential hidden variables, we performed experiments exploring how hidden variables expand the power of Bayesian networks. The explorations by Rusakov and Geiger [14; 48] focused on deriving new scoring functions to evaluate the usefulness of hidden variables in a network. We instead pursue an investigation of when hidden variables add representational power to the Bayesian network. This investigation leads to a more concrete understanding of how Bayesian networks with hidden variables can represent a wider variety of distributions.

6.3.1 Overview of the Algorithm

Our algorithm explores the space of graphs with essential hidden variables based on the idea that a Bayesian network B of size n with one of its attributes hidden is sometimes

[§] *The summation operator in this context denotes the removal of the variable summed over by adding up the probability of then normalizing over all of that attribute's values (i.e.* $\sum_B$ *means "sum out* B *from these probability calculations").*

$1 \perp\!\!\!\perp H$	$1 \perp\!\!\!\perp H \mid 3$	$1 \perp\!\!\!\perp H \mid 4$	$1 \perp\!\!\!\perp H \mid \{3, 4\}$
$1 \perp\!\!\!\perp 3 \mid H$	$1 \perp\!\!\!\perp 3 \mid 4$	$1 \perp\!\!\!\perp 3$	$1 \perp\!\!\!\perp 3 \mid \{H, 4\}$
$1 \perp\!\!\!\perp 4$	$1 \perp\!\!\!\perp 4 \mid 2$	$1 \perp\!\!\!\perp 4 \mid 3$	$1 \perp\!\!\!\perp 4 \mid H$
$1 \perp\!\!\!\perp 4 \mid \{2, H\}$	$1 \perp\!\!\!\perp 4 \mid \{3, H\}$	$1 \perp\!\!\!\perp 4 \mid \{2, 3, H\}$	
$2 \perp\!\!\!\perp 4$	$2 \perp\!\!\!\perp 4 \mid H$	$2 \perp\!\!\!\perp 4 \mid 1$	$2 \perp\!\!\!\perp 4 \mid \{H, 3\}$
$2 \perp\!\!\!\perp 4 \mid \{1, H\}$	$2 \perp\!\!\!\perp 4 \mid \{1, H, 3\}$		
$H \perp\!\!\!\perp 4$	$H \perp\!\!\!\perp 4 \mid 1$	$H \perp\!\!\!\perp 4 \mid 2$	$H \perp\!\!\!\perp 4 \mid \{1, 2\}$

Table 6.2 Complete list of independences in the Bayesian network appearing in Figure 6.3.

more powerful than any Bayesian network of size $n - 1$. This happens when B represents an underlying distribution that no network of size $n - 1$ can represent—the essential hidden variable then has an impact on the visible attributes that cannot be duplicated with $n - 1$ attributes (and no hidden variable).

Given a Bayesian network B of size n (using as an example Figure 6.3, the W-network, which is a network of size 5), the algorithm proceeds as follows: we assume that B is faithful to the domain of interest so D-separation rules are applied to generate the complete set of independences I_B present in B. We then choose a single attribute H to hide by removing all independences in I_B referring directly to H. Note that, crucially, some independences generated by the d-separation rules do not directly refer to H but nonetheless rely on H's presence and these are left in I_B. Thus, while no independence is allowed to include H, the d-separation rules may allow H to influence the independences B represents.

For the network in Figure 6.3, Table 6.2 contains the complete list of independences. The list with independences not directly referring to H is $I_B = \{(1 \perp\!\!\!\perp 3), (1 \perp\!\!\!\perp 4), (1 \perp\!\!\!\perp 3 \mid 4), (2 \perp\!\!\!\perp 4), (1 \perp\!\!\!\perp 4 \mid 2), (1 \perp\!\!\!\perp 4 \mid 3), (2 \perp\!\!\!\perp 4 \mid 1)\}$.

We then generate all Bayesian networks of size $n - 1$ (referred to collectively as the "smaller networks") and compare the sets of independences represented by each smaller network to I_B: if the two sets do not match exactly, we continue to the next network of size

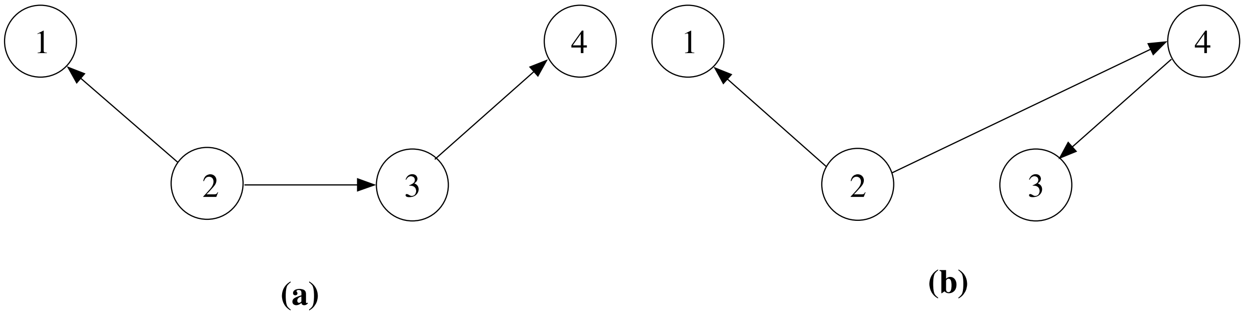


Figure 6.5 Example networks of size 4.

$n-1$. If the set of independences generated by some smaller network and I_B match exactly, we conclude that H is not an essential hidden variable and try hiding a different attribute as a possible hidden variable in B. If we examine all networks of size $n-1$ without finding a smaller network that generates exactly I_B, we can conclude by exhaustion that H is an essential hidden variable because any probability distribution generating I_B requires H to generate a faithful Bayesian network.

Continuing our example of the W-network, consider the networks of size 4 depicted in Figure 6.5. Amongst other differences between the independence sets of these two networks and the W-network (Figure 6.3), the network in Figure 6.5(a) does not have independence $(2 \perp\!\!\!\perp 4)$ while Figure 6.5(b) has the extra independence $(2 \perp\!\!\!\perp 3 \mid 4)$. Thus these two networks are rejected as matches for the distribution the W-network represents. The algorithm then continues to examine all networks of size 4 to match the independences generated by each network to I_B. In this example, I_B is not generated by any network of size 4.

Algorithm 5 presents the pseudocode that generalizes this search procedure to examining all networks of size n for hidden variables.

6.3.2 Optimizations

The main problem with Algorithm 5 is that it runs in exponential time—the number of possible directed, acyclic graphs over n variables $f(n)$ is characterized by the recursion

Algorithm 5 Basic Hidden Variable Detection Algorithm.

```
for each network $B$ of size $n$ do
    $L = \emptyset$
    Generate the set of independences in $B$, $I$
    for each attribute $h$ in the network $B$ do
        Remove any independences mentioning of $h$ from $I$ to generate $I_h$
        for each network $B'$ of size $n-1$ do
            Generate the independences in $B'$, $I'$
            if $I_h == I'$ then
                $h$ is not an essential hidden variable, break and try another $h$
            end if
            Add $(h, B)$ to $L$
        end for
    end for
    Output $L$
end for
```

$f(n) = \sum_{i=1}^{n}(-1)^{i+1}\binom{n}{i}2^{i(n-i)}f(n-i)$ [47] and, asymptotically, $f(n) \in O(2^{n^2-2})$. The log of this function appears in Figure 6.6 as the solid curve.

We now discuss a few optimizations that leads to larger networks being processed more efficiently. The major optimizations of memorizing independences, graph isomorphism usage, and independence isomorphism usage are discussed below.[¶] After all the optimizations are explained, a revised version of Algorithm 5 including optimization detail is presented.

6.3.2.1 Memorizing independences

The simplest approach to optimizing the run time of any algorithm with repetition is to expand the amount of space used by the algorithm: by memorizing independence information about all of the networks of size $n-1$, the innermost "for" loop of Algorithm 5 can be executed very quickly for each network of size n. However, this is only a partial solution, as the independences of the complete set of networks of size 7 or larger cannot be held

[¶] A fourth optimization, testing for connected components in the graph, was found to result in no efficiency gain [42] so it has been omitted from this discussion.

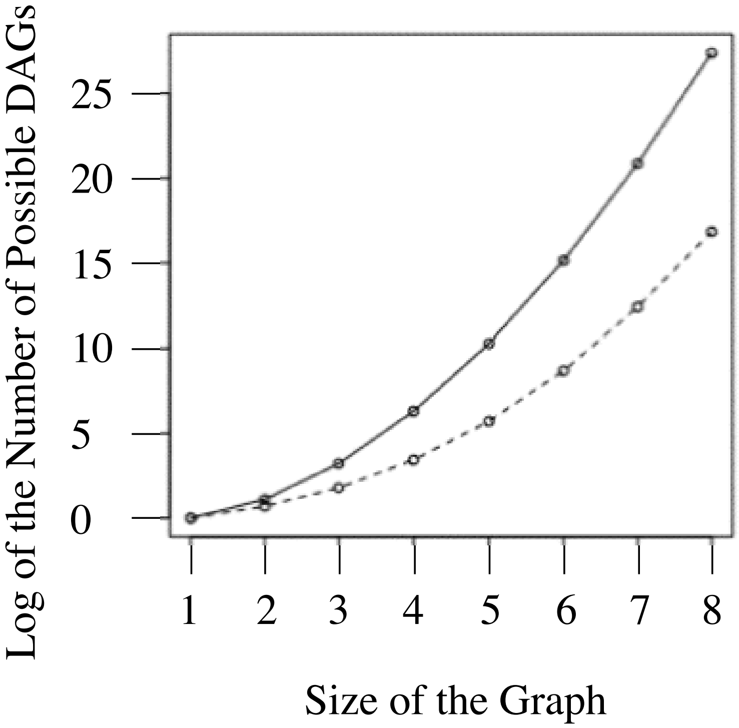


Figure 6.6 Graph of the size of any directed, acyclic graph (DAG) against the log of the number of possible DAGs (solid line) and number of non-isomorphic possible DAGs (dotted line).

in memory, requiring that we instead memorize as many as we can then recalculate the independences corresponding to the remaining smaller networks for each large network.

6.3.2.2 Graph Isomorphism Equivalence Classes

A more fundamental alteration to Algorithm 5 is to restrict ourselves to testing Bayesian networks that are not graph isomorphic to any previously-examined Bayesian network. Algorithm 5 tests several networks in the same graph isomorphism class. For example, if attribute 0 was discovered to be a hidden variable in a 5 vertex graph (as in the W-network, Figure 6.3), attribute 1 would be a hidden variable in the network where the identities of 0 and 1 are switched, attribute 2 when 0 and 2 are switched, etc. To illustrate the difference in the number of graphs that need to be tested, Figure 6.6 presents the logarithm of the total number of directed acyclic graphs of each size (from [47]) with the number of isomorphism classes (from [33]).

By only testing one graph from each isomorphism class, we can save a significant amount time in both loops of Algorithm 5—the search space of both large and small networks is greatly reduced by examining only graph non-isomorphs. Unfortunately, this reduction in the search space is only feasible up to graphs of size 8 because the problem of efficiently discovering all graph isomorphism classes is slow—it is suspected to not be in P [35]. Also, while the optimization results in an increasing reduction in the number of networks tested as number of attributes increases, isomorphism still only results in a small reduction in number of graphs. As we can see from Figure 6.6, the number of non-isomorphic graphs of size n is approximately the total number of graphs of size $n - 1$.

The algorithm generates graph isomorphisms by reading graphs of size 8 or less from files generated previously by McKay's NAUTY algorithm [34]. NAUTY is acknowledged as the fastest overall graph automorphism and isomorphism detection algorithm with tight run time bounds of $O(n^2)$ and $\Omega(2^n)$ for the best and worst case (respectively) of processing a graph of size n.[‖]

With the testing of only non-graph-isomorphic Bayesian networks, it became necessary to be able to determine if two sets of independences are isomorphic. The labeling used in the larger network and the labeling used in the smaller network may differ but the independences could be the same when labels are permuted differently.

Two sets of independences (one from the larger network and one from a smaller network) are tested for equivalence in the following way:

1. Verify that each set has the same cardinality.

2. See if the independences match without altering any labels.

3. Make sure the same number of attributes are in evidence for each corresponding independence. For example, if one independence set has 3 independences and each independence has 2 attributes in evidence, the other independence set must have 3 independences with 2 attributes in evidence also.

4. Recursively try every possible relabeling of vertices in one independence set to match the other set. Although this brute force algorithm takes $O(n!)$ time to verify there is no such relabeling, the probability that two non-matching sets will make it to this step is low so, in practice, this step rarely needs to be executed and, if run, usually results in quickly finding a mismatch.

Algorithm 6 incorporates all of the above optimizations.

6.3.2.3 Neapolitan's DetermineFaithful

Taking a different approach from the previous optimizations, instead of checking each network B' of size $n-1$, we use DETERMINEFAITHFUL to discover if there is a B' of size

[‖] See [35], Theorems 5.2 and 6.2, for details.

```
Seen = $\emptyset$
Seen$_{sm}$ = $\emptyset$
for each network $B$ of size $n$ not isomorphic to a graph in Seen do
  Add $B$ to Seen
  $L = \emptyset$
  Generate the set of independences in $B$, $I$
  for each attribute $h$ in the network $B$ do
    Remove any independences mentioning of $h$ from $I$ to generate $I_h$
    for each network $B'$ of size $n - 1$ not isomorphic to a graph in Seen$_{sm}$ do
      Add $B'$ to Seen$_{sm}$
      Generate the independences in $B'$, $I'$
      if $I_h == I'$ then
        $h$ is not an essential hidden variable, break and try another $h$
      end if
      Add $(h, B)$ to $L$
    end for
  end for
  Output $L$
end for
```

Algorithm 6   Algorithm for Hidden Variable Detection with Optimizations.

$n-1$ that is faithful to the distribution represented by I_h. This will result in the same set of conclusions [39] but it is not clear whether this method will be any faster than Algorithm 6. This approach is implemented as Algorithm 7.

```
Seen = $\emptyset$
Seen$_{sm}$ = $\emptyset$
for each network $B$ of size $n$ not isomorphic to a graph in Seen do
   Add $B$ to Seen
   $L = \emptyset$
   Generate the set of independences in $B$, $I$
   for each attribute $h$ in the network $B$ do
      Remove any independences mentioning of $h$ from $I$ to generate $I_h$
      if DetermineFaithful is successful with input $I_h$ then
         $h$ is not an essential hidden variable, break and try another $h$
      end if
      Add $(h, B)$ to $L$
   end for
   Output $L$
end for
```

Algorithm 7 Algorithm for Hidden Variable Detection with DetermineFaithful.

6.3.3 Experimental Results

The goal of the experiments is to discover what conditions will always hold around essential hidden variables by analyzing the set of Bayesian network and hidden variable pairs returned by Algorithm 6. If a set of edge or independence constraints holds in the neighborhood of every essential hidden variable, future work may be able to prove these constraints holds in all distributions, implying an essential hidden variable. Other algorithms may then be able to exploit these experimentally verified constraints to find essential hidden variables through sets of local tests.

The algorithms were implemented in Java 1.4.2 on a two-processor 2.8 Ghz Xeon computer with 2 Gb of RAM.

6.3.3.1 Edge Test Results

Tests were done to find out if the edges of the graph around the essential hidden variable conformed to any set of edge constraints. The constraints sought required edges to or from the essential hidden variable, to or from attributes near the hidden variable, and/or required some edges to not exist in the graph. We used the graph isomorphism subprocedures to find out if particular edge sets held in every graph with an essential hidden variable.

It was found that the subset of the edge constraints given in the definition of W-networks (Definition 6.2.9) held around all essential hidden variables. This means that the W-network is always found embedded in a network with an essential hidden variable and specifically that the hidden variable was always at the apex of the middle peak in the "W" of the W-network.

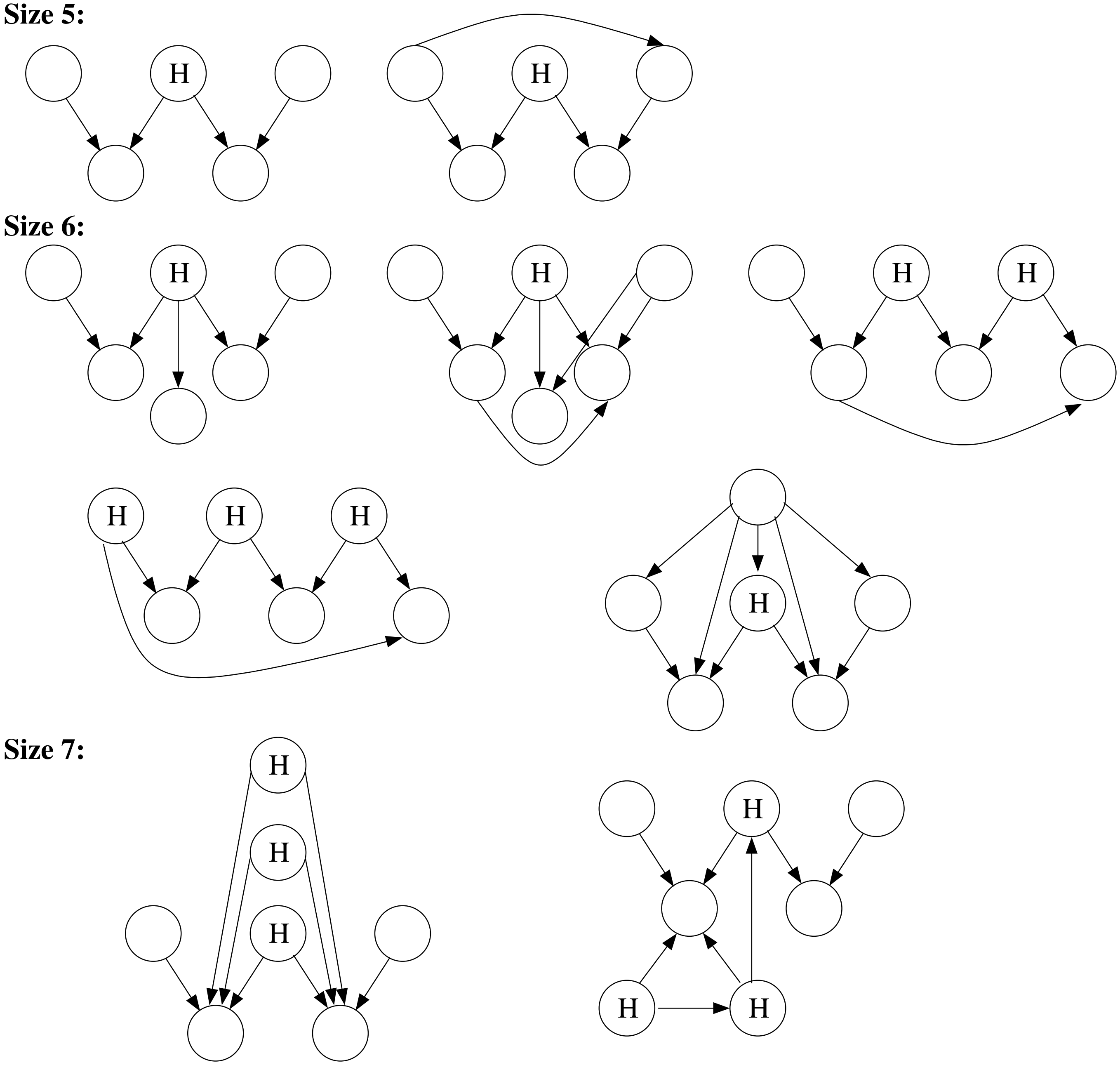
6.3.3.2 Independence Test Results

Various sets of independence constraints were given to the algorithm to test against each network with an essential hidden variable. These were given as independences or dependencies between attributes—the hidden variable was not included in these tested properties in order to produce results that generalize more easily to an empirical test.

A set of independence constraints that we attempted to verify was $(1 \perp\!\!\!\perp 3 \mid 4)$, $(1 \not\perp\!\!\!\perp 3 \mid 2)$, $(1 \not\perp\!\!\!\perp \{2, 4\})$, and $(2 \perp\!\!\!\perp 4 \mid 1)$. This set held in both networks of size 5 that had an essential hidden variable (see Fig. 6.7 below) but did not hold in all networks of size 6 or higher with an essential hidden variable. In fact, no set of independence constraints that we tried held in networks of size 6 or larger. This indicates that it may not be possible to characterize essential hidden variables through independences among the visible attributes.

6.3.3.3 Summary and Examples

Table 6.3 displays the full count of all networks with hidden variables as well as average run times for Algorithms 5, 6, and 7. Some algorithm runs could not be completed in 36



Size 5:

Size 6:

Size 7:

Figure 6.7 Examples of Bayesian networks with essential hidden variables of size 5, 6, and 7. Each node marked with an H was drawn separately from the network—multiple hidden variables were not tested simultaneously.

Table 6.3 Average Running Time of EHV Detection Algorithms (seconds).

Size	Alg	Num Nets	Num HVs	Median Time	Avg Time	SD
4	5	543	0	0	0	0.001
4	6	31	0	0.001	0.002	0.003
4	7	31	0	0.001	0.001	0.002
5	5	29280	100	0.038	0.04	0.016
5	6	303	2	0.001	0.001	0.001
5	7	303	2	0.001	0.002	0.001
6	5	> 54161	> 2185	1.731	2.392	1.922
6	6	5985	156	0.008	0.01	0.008
6	7	5985	156	0.005	0.006	0.002
7	5	> 800	> 67	7.542	162.010	397.395
7	6	> 41826	> 3815	2.819	3.098	2.035
7	7	243669	16103	0.019	0.021	0.009
8	5	> 0	> 0	0	0	0
8	6	> 39222	> 3710	2.838	3.151	2.106
8	7	> 2450470	> 403852	0.053	0.054	0.015

hours (indicated by the $> X$ notation for sizes; time data from the last time output was given in the last 3 columns in Table 6.3). Note that the results for incomplete analysis will be biased towards smaller, simpler graphs on average (as graphs with fewer edges are tested first) so the timing information will be slightly lower than true values. Performance on graphs of size 9 and higher is not reported as preparation to process these graphs took longer than the time allowed (over 36 hours). Test times for Algorithms 6 and 7 do not account for the time spent to generate the isomorphism classes of graphs — this was 0.08 seconds for size 6, 2.2 seconds for size 7, and 260 seconds for size 8 (for smaller graphs, time was negligable and, for larger graphs, over a day was taken). Note in particular that Algorithm 7 is faster than Algorithm 6 so DetermineFaithful [39] is recommended for future work in this area.

Some examples of graphs discovered with essential hidden variables of size 5, 6, and 7 are given in Figure 6.7. Recall that each hidden variable discovered in a network of size n is discovered predicated on the other $n - 1$ attributes being visible. What appears to indicate multiple hidden variables in fact indicates that the other hidden variables are added back to the network as visible attributes before the next hidden variable is tested.

6.3.4 Integration of Results with Previous Research

The algorithm we devised to discover essential hidden variable resembles that of the method of vanishing tetrad differences [55] in that we will examine independences between the visible attributes to derive essential hidden variables. It differs in that our methods do not rely on the variables being continuous or in a linear relationship to one another but is limited to essential hidden variables discoverable through independence relations.

As no independence-based constraints on the distribution containing an essential hidden variable were found, the non-independence characterization discussed in Verma's P Network (Definition 6.2.10) might provide another set of probabilistic properties to characterize essential hidden variables. However, it is not clear how these constraints would generalize for generating essential hidden variables in larger networks. A first step to finding this relationship would be equipping the current algorithm with the ability to test non-independence-based constraints.

Models with fewer than two links between hidden and measured attributes must introduce a correction factor (based on the number of measured attributes) to avoid asymptotic divergence of the optimizing score [48]. As the W-Network has exactly two links between the hidden variable and the visible attributes, this points to a possible connection of essential hidden variables to networks that were found cause score evalution algorithms to not converge [48].

The algorithm proposed in [39] was tested indirectly via the DetermineFaithful algorithm being used as a subprocedure in Algorithm 7. While all algorithms ran quickly for small sizes, Algorithm 7 was the fastest while still generating equivalent results to the other algorithms. However, we should note that the algorithms proposed to find causality [39] (especially while searching for embedded independences) are not as flexible as the algorithms proposed here. For example, it would be difficult to use those algorithms to find interactions between hidden variables.

6.3.5 Conclusions Based on the EHV Detection Algorithms

Our experiments show that networks of the sizes examined all contain a subset of the edges and non-edges present in a W-network embedded around the hidden variable. Precisely all the edges that are and are not allowed and what this implies for the independences between measured attributes has yet to be determined. However, if examining all the networks of a certain size is required, significant optimization will be needed to answer these questions in a reasonable time frame.

APPENDIX A. MRCA Simulator

This appendix is simply a copy of commented Java 1.6 code for a MRCA simulator. Code can be found at http://www.cs.iastate.edu/ patterbj/diss.php#simulator .

APPENDIX B. MRCA Construction Rules for $Y > X^2$

A complete listing of all the rules needed to MRCA compute the region $X = [(x, y) \in [0, 1]^2 \mid y > x^2]$ can be found at http://www.cs.iastate.edu/ patterbj/diss.php#construction

.

APPENDIX C. Computation of In-Place MRCA Rules for Rational Lines

The code found at http://www.cs.iastate.edu/ patterbj/diss.php#inplace constructs a graphical user interface for computing rules sets to color any region delimited by a rational line. This code uses the code given in Appendix A.

BIBLIOGRAPHY

[1] N. A., "Linear automaton transformation," *Proceedings of the American Mathematical Society*, vol. 9, pp. 541–544, 1958.

[2] P. G. Bergmann, *Introduction to the Theory of Relativity.* Dover Publications, 1976.

[3] J. Binder, D. Koller, S. Russell, and K. Kanazawa, "Adaptive probability networks with hidden variables," *Machine Learning*, vol. 29, pp. 213–244, 1997.

[4] V. Brattka and K. Weihrauch, "Computability on subsets of Euclidean space I: Closed and compact subsets," *Theoretical Computer Science*, vol. 219, pp. 65–93, 1999.

[5] M. Braverman, "On the complexity of real functions," in *Forty-Sixth Annual IEEE Symposium on Foundations of Computer Science*, 2005, pp. 155–164.

[6] M. Braverman and S. Cook, "Computing over the reals: Foundations for scientific computing," *Notices of the AMS*, vol. 53, no. 3, pp. 318–329, 2006.

[7] A. Burks, *Essays on Cellular Automata.* Univeristy of Illinois Press, 1970.

[8] L. Carroll, *Alice's Adventures in Wonderland.* Project Gutenberg, 2008. [Online]. Available: http://www.gutenberg.org/ebooks/11

[9] G. J. Chaitin, "On the length of programs for computing finite binary sequences: statistical considerations," *Journal of the ACM*, vol. 16, pp. 145–159, 1969.

[10] ——, “On the number of n-bit strings with maximum complexity,” *Applied Mathematics and Computation*, vol. 59, pp. 97–100, 1993.

[11] B. Copeland, “Accelerating Turing machines,” *Minds and Machines*, vol. 12, pp. 281–301, 2002.

[12] A. Ehrenfeucht, R. Parikh, and G. Rozenberg, “Pumping lemmas for regular sets,” *SIAM Journal of Computing*, vol. 10, pp. 536–541, 1981.

[13] A. Einstein, *Relativity: The Special and General Theory.* New York, NY: H. Holt and Company, 1916.

[14] D. Geiger, D. Heckerman, H. King, and C. Meek, “Stratified exponential families: Graphical models and model selection,” Microsoft Research, Technical Report MSR-TR-98-31, 1998.

[15] D. Geiger and C. Meek, “Graphical models and exponential families,” in *Proceedings of the 14th Conference on Uncertainty in AI*, 1998, also Microsoft Tech Report MSR-TR-98-10.

[16] D. Griffeath and C. Moore, *New Constructions in Cellular Automata.* USA: Oxford University Press, 2003.

[17] A. Grzegorczyk, “Computable functionals,” *Fundamenta Mathematicae*, vol. 42, pp. 168–202, 1955.

[18] J. D. Hamkins and A. Lewis, “Infinite time Turing machines,” *Journal of Symbolic Logic*, vol. 65, no. 2, pp. 567–604, 2000.

[19] M. J., “Finite automata and the representation of events,” Wright Patterson AFB, Dayton, Ohio, Tech. Rep. 57-624, 1957.

[20] J. Jaffe, “A neccessary and sufficient pumping lemma for regular languages,” *SIGACT News*, vol. 10, pp. 48–49, 1978.

[21] R. Kiester and K. Sahr, “Planar and spherical hierarchical multi-resolution cellular automata,” *Computers, Environment, and Urban Systems*, vol. 32, no. 3, pp. 204–213, 2008.

[22] K. Ko and H. Friedman, “Computational complexity of real functions,” *Theoretical Computer Science*, vol. 20, pp. 323–352, 1982.

[23] K.-I. Ko, *Complexity Theory of Real Functions.* Boston: Birkhäuser, 1991.

[24] A. N. Kolmogorov, “Three approaches to the quantitative definition of ‘information’,” *Problems of Information Transmission*, vol. 1, pp. 1–7, 1965.

[25] D. Kozen, *Automata and Computability.* New York, NY: Springer-Verlag, 1997.

[26] C. Kreitz and K. Weihrauch, “Complexity theory on real numbers and functions,” in *Theoretical Computer Science*, ser. Lecture Notes in Computer Science, vol. 145. Springer, 1982, pp. 165–174.

[27] D. Lacombe, “Extension de la notion de fonction recursive aux fonctions d’une ow plusiers variables reelles, and other notes,” *Comptes Rendus*, 1955, 240:2478-2480; 241:13-14, 151-153, 1250-1252.

[28] J. Lathrop, J. Lutz, and B. Patterson, “Multi-resolution cellular automata for real computation,” in *Proceedings of Computability in Europe 2011*, 2011.

[29] M. Li and P. Vitányi, *An Introduction to Kolmogorov Complexity and Its Applications*, 3^{rd} ed. New York, NY: Spring Science + Business Media, LLC, 2008.

[30] L. R. Lisagor, “The banach-mazur game,” *translated in Math. USSR Sbornik*, vol. 38, pp. 201–206, 1981.

[31] J. H. Lutz, “Category and measure in complexity classes,” *SIAM Journal on Computing*, vol. 19, pp. 1100–1131, 1990.

[32] H. V. McIntosh, *One Dimensional Cellular Automata.* United Kingdom: Luniver Press, 2009.

[33] B. McKay, “Combinatorial data. http://cs.anu.edu.au/people/bdm/data/digraphs.html,” p. http://cs.anu.edu.au/people/bdm/data/digraphs.html, 2004.

[34] ——, “The nauty page. http://cs.anu.edu.au/people/bdm/nauty/,” p. http://cs.anu.edu.au/people/bdm/nauty/, 2004.

[35] T. Miyazaki, “The complexity of mckay’s canonical labeling algorithm,” in *Conference on Discrete Mathematics and Computer Science (DIMACS)*, Einkelstien and Kantor, Eds., vol. 28, 1997.

[36] W. Myrvold, “The decision problem for entanglement,” in *Potentiality, Entanglement and Passion-at-a-Distance: Quantum Mechanical Studies for Abner Shimony*, M. S. J. Cohen, R.S.; Horne, Ed. Dordrecht and Boston: Kluwer Academic Publishers, 1997, pp. 177–190.

[37] K. Nagel and M. Schreckenberg, “A cellular automaton model for freeway traffic,” *Journal de Physique I*, vol. 2, no. 12, pp. 2221–2229, 1992.

[38] K. Nagel, P. Stretz, M. Pieck, S. Leckey, R. Donnelly, and C. Barrett, “Transims traffic flow characteristics,” Los Alamos Unclassified Report, Los Alamos National Laboratory, Tech. Rep. 97-3530, 1997.

[39] R. Neapolitan, *Learning Bayesian Networks.* Prentice Hall, 2003.

[40] M. W. Parker, “Undecidability in $\mathbb{R}^n$: Riddled basins, the KAM tori, and the stability of the solar system,” *Philosophy of Science*, vol. 70, pp. 359–382, 2003.

[41] ——, “Three concepts of decidability for general subsets of uncountable spaces,” *Theoretical Computer Science*, vol. 351, pp. 2–13, 2006.

[42] B. Patterson, “Essential hidden variables: An introduction and novel algorithm for detection,” Masters of Science, Iowa State University, 2004.

[43] B. Patterson and D. Margaritis, “Essential hidden variables: An exploratory algorithm,” in *Proceedings of the 2nd Indian International Conference on Artificial Intelligence*, 2005, pp. 2649–2667.

[44] J. Pearl, “A constraint-propagation approach to probabilistic reasoning,” in *Proceedings of the 2nd Conference on Uncertainty in AI*, 1986, pp. 357–370.

[45] ——, *Probabilistic Reasoning in Intelligent Systems*, 2nd ed. San Francisco, CA: Morgan Kaufmann Publishers, Inc., 1988.

[46] M. B. Pour-El and J. I. Richards, *Computability in Analysis and Physics.* Springer-Verlag, 1989.

[47] R. W. Robinson, “Counting labelled acyclic graphs,” in *New Directions in the Theory of Graphs*, F. Harary, Ed. New York, New York: Academic Press, Inc., 1971, pp. 239–273.

[48] D. Rusakov and D. Geiger, “Asymptotic model selection for naive bayesian networks,” in *Proceedings of the 18th Conference on Uncertainty in AI*, 2002, pp. 438–445.

[49] K. Sahr, D. White, and A. Kimerling, “Geodesic discrete global grid systems,” *Cartography and Geographic Information Science*, vol. 30, no. 2, p. 121134, 2003.

[50] M. Schaller and K. Svozil, “Scale-invariant cellular automata and self-similar Petri nets,” *The European Physical Journal B*, vol. 69, p. 297311, 2009. [Online]. Available: http://dx.doi.org/10.1140/epjb/e2009-00147-x

[51] J. Schiff, *Cellular Automata: A Discrete View of the World.* USA: Wiley-Interscience, 2008.

[52] H. A. Simon, "Spurious correlation: A causal interpretation," *Journal of the American Statistics Association*, vol. 49, no. 267, pp. 467–479, 1954.

[53] R. J. Solomonoff, "A formal theory of inductive inference," *Information and Control*, vol. 7, pp. 1–22, 224–254, 1964.

[54] P. V. Spade, "Ockham's razor. http://plato.stanford.edu/archives/fall2002/entries/ockham/#4.1," 2002.

[55] P. Spirtes, C. Glymour, and R. Scheines, *Causation, Prediction, and Search*, 2nd ed. Cambridge, MA: The MIT Press, 2000.

[56] D. Stanat and S. Weiss, "A pumping theorem for regular languages," *SIGACT News*, vol. 14, pp. 36–37, 1982.

[57] A. M. Turing, "On computable numbers with an application to the Entscheidungsproblem," *Proc. London Math. Soc. (2)*, vol. 42, pp. 230–265, 1936.

[58] ——, "On computable numbers with an application to the Entscheidungsproblem. A correction," *Proc. London Math. Soc. (2)*, vol. 43, pp. 544–546, 1936.

[59] ——, "Systems of logic based on ordinals," Ph.D. dissertation, Princeton, Princeton, New Jersey, USA, 1938.

[60] S. Varricchio, "A pumping condition for regular sets," *SIAM Journal of Computing*, vol. 26, pp. 764–771, 1997.

[61] K. Weihrauch, *Computability.* New York, NY, USA: Springer-Verlag New York, Inc., 1987.

[62] ——, *Computable Analysis. An Introduction.* Springer-Verlag, 2000.

[63] S. Wolfram, *A New Kind of Science.* Champaign, Illinois, USA: Wolfram Media, 2002.