Algorithmic Construction and Stochastic Analysis of Optimal Automata for Generalized Strings

by

Ian Guo-fan Char

A thesis submitted to the

Faculty of the Graduate School of the

University of Colorado in partial fulfillment

of the requirements for the degree of

Master of Science

Department of Applied Mathematics

2018

This thesis entitled:
Algorithmic Construction and Stochastic Analysis of Optimal Automata for Generalized Strings
written by Ian Guo-fan Char
has been approved for the Department of Applied Mathematics

_____

Prof. Manuel E. Lladser

_____

Prof. Jem Corcoran

_____

Dr. Anne Dougherty

Date _____

The final copy of this thesis has been examined by the signatories, and we find that both the content and the form meet acceptable presentation standards of scholarly work in the above mentioned discipline.

Ian Guo-fan Char, (M.S., Applied Mathematics)

Algorithmic Construction and Stochastic Analysis of Optimal Automata for Generalized Strings

Thesis directed by Prof. Manuel E. Lladser

In many applications, the need arises to search a text for appearances of a given set of keywords. As an example, in bioinformatics one may wish to search a DNA sequence to find so-called *biological motifs*. A standard approach to this problem is to leverage a *deterministic finite automaton*—a graph structure which is traversed as letters of the text are read in. However, depending on the number and length of the keywords being sought in the text, the graph may be too large to fit in computer memory, making this approach fruitless.

In this thesis, we first present a novel algorithm that, under the assumption that the keywords take the form of a so-called *generalized string*, constructs the minimal DFA recognizing those keywords. Importantly, the algorithm is iterative and allows one to build the automaton directly, without any use of buffer memory. Not only does this mean that the algorithm is efficient regarding memory consumption, but it also provides useful insight to help facilitate analysis for the size of such DFA.

Pairing this new algorithm with the assumption that the generalized strings are drawn at random from some class of probability distributions, we develop bounds on the size of the minimal automaton that are true with high probability. Furthermore, using synthetic data, we provide evidence that the size of the minimal automaton grows linearly in expectation for many cases.

Dedication

To my loving parents for giving their full support regardless of where my passions lead me.

Acknowledgements

To start, I would like to thank my advisor, Manuel Lladser, for his guidance and support throughout the project. He was the one that introduced me to this subject area, and it was his expertise and inspiration that guided me through this process. This thesis was made possible largely through his help.

Furthermore, I would like to thank the other members of my committee, Jem Corcoran, and Anne Dougherty. It was through their courses that I was able to develop a lot of the technical skills that I needed to complete this thesis. In particular, I would like to thank Anne Dougherty for advising me throughout my undergraduate career as well as funding this research through the National Science Foundation EXTREEMS grant, DMS 1407340.

Lastly, I would like to thank my family for their support throughout this process.

Contents

Chapter

Tables

Table

Figures

Figure

Chapter 1

Introduction and Overview

In the first chapter of this thesis, we briefly give motivation for studying minimal automata as they relate to generalized strings. Furthermore, preliminaries covering the notion of *languages* and *deterministic finite automata* are given in this chapter. We end the chapter by giving an outline of the rest of the thesis.

1.1 Motivation

An everyday use of computers is to find all locations in which given keywords appear within a text. A ubiquitous example of this is the "find" feature implemented in internet browsers, which allows the user to search a web page for a particular word. Searches may, however, significantly grow in complexity in other contexts.

For example, CRISPR is a technique for genetic modification that has recently garnered a lot of attention. A caveat to this approach is that genetic modifications can only be made next to specific DNA sequences called *proto-spacer adjacent motifs* (PAMs). It is therefore imperative to be able to locate PAMs within a DNA strand that is possibly composed of several million bases. This is one of many cases where finding so-called *biological motifs* within a DNA or RNA sequence is beneficial.

A possible way to tackle this problem would be to construct a *deterministic finite automaton* (DFA) to search for the motif. Broadly speaking, this is a graph-based structure that allows one to keep track of what has been seen while scanning through the text. A major obstacle to employing

such a DFA, however, is the possibility that it will consume too much memory, making it infeasible to use.

In the remainder of the chapter, we explore the concepts of languages and DFA, state their formal definitions, and show how memory consumption can be a major obstacle when tackling certain motifs.

1.2 Languages

Define an alphabet, represented by Σ, to be the set of possible letters (or symbols) that can appear in a text. In the same way that one writes words in English, one can use the letters in an alphabet to construct a *string*. For example, given the alphabet $\Sigma = \{A, B, C, D\}$, one can construct the strings CAD, $AAAAAA$, and $BDAACC$. Regardless of the alphabet, one possible string is the *empty string*, which is denoted by ε and contains no letters. (Note that, by assumption, $\varepsilon \notin \Sigma$.)

In general, one can represent a string by concatenating letters from the alphabet. That is, if $\sigma_1, \sigma_2, \ldots, \sigma_\ell \in \Sigma$ then $s = \sigma_1\sigma_2 \ldots \sigma_\ell$ represents a string. In this case, we say that s has *length* ℓ and write $|s| = \ell$. (By definition, $|\varepsilon| = 0$.) Furthermore, $s[i]$ denotes the i^{th} letter that appears in the string s; in particular, $s[i] = \sigma_i$ for $1 \leq i \leq \ell$. This concept can be extended to substrings contained in s. In particular, for $i \leq j \leq \ell$, denote $s[i : j] = \sigma_i\sigma_{i+1} \ldots \sigma_j$. For simplicity, $s[: j] \coloneqq s[1 : j]$ and $s[i :] \coloneqq s[i : |s|]$. (Note that $s[: j] = \varepsilon$ for $j \leq 0$, and $s[i :] = \varepsilon$ for $i > |s|$.) Furthermore, given two strings $s_1 = \sigma_1\sigma_2 \ldots \sigma_{\ell_1}$ and $s_2 = \hat{\sigma}_1\hat{\sigma}_2 \ldots \hat{\sigma}_{\ell_2}$, one can *concatenate* them to create a new string. That is, $s_1 s_2 \coloneqq \sigma_1\sigma_2 \ldots \sigma_{\ell_1}\hat{\sigma}_1\hat{\sigma}_2 \ldots \hat{\sigma}_{\ell_2}$.

In what follows, the set of all possible finite length strings that can be composed using the alphabet Σ is written as Σ^*. We also write Σ^+ to denote the set all finite, positive length strings; in particular, $\Sigma^+ \coloneqq \Sigma^* \setminus \{\varepsilon\}$. A *language* with respect to a reference alphabet Σ is any non-empty subset of Σ^*. Thus, for instance, Σ^+ is a language.

Like strings, languages can be concatenated together to create new languages. Namely, given languages $\mathcal{A}$ and $\mathcal{B}$, we define $\mathcal{AB} = \{ab : a \in \mathcal{A}, b \in \mathcal{B}\}$. For example, if $\mathcal{A} = \{CD, CAT\}$ and

$\mathcal{B} = \{BAT, DOG\}$, then $\mathcal{AB} = \{CDBAT, CDDOG, CATBAT, CATDOG\}$. In many cases, where s is a string and L is a language, we write sL or Ls as short hand for $\{s\}L$ and $L\{s\}$, respectively.

Returning to the originally posed problem about finding keywords within a text, we can now think of this as identifying prefixes of the text that belong to the language $\Sigma^* L$, where L is the set of keywords of interest. This is because $\Sigma^* L$ is the set of all strings that have a suffix belonging to L.

To fix ideas, let Σ be the regular English alphabet with spaces, let $L = \{cat, dog\}$, and let the text to be analyzed be: "the cat jumped over the dog". If we are looking for strings in $\Sigma^* L$ and we are reading the text from left to right, we will recognize the strings "the cat" and the full text "the cat jumped over the dog" as belonging to this language. Moreover, since the suffixes of these strings must contain either "cat" or "dog", their lengths allow us to infer the location within the text at which instances of L occur.

1.3 Deterministic Finite Automata

In this section, we will briefly introduce the concept of a *deterministic finite automaton*. To learn more about this topic, the reader is encouraged to read Hopcroft, Motwani, and Ullman's *Introduction to Automata Theory, Languages and Computation* [5].

At a high level, a deterministic finite automaton (DFA, used for both plural and singular) is a finite, directed graph with edges labelled with letters from the alphabet. It has a distinguished vertex called the *initial state*. Starting at this state, and as characters in a text are read, transitions between states happen depending on the next letter read in from that text. Possibly several of the states in the automaton are marked as *terminal states*. An arrival to one of these states implies that a prefix of text in the language of interest has been encountered. One major restriction to DFA is that they can only recognize *regular languages*. However, in our setting, we need not worry about this because we assume L to be finite (and hence regular), and thus $\Sigma^* L$ is also regular.

The following defines a DFA more precisely.

Definition 1.1. *A DFA is a 5-tuple of the form $M = (Q, \Sigma, \delta, q_0, F)$, where Q is a finite set of elements called states, Σ is the reference alphabet associated with the DFA, $\delta : Q \times \Sigma \to Q$ is the so-called transition function, $q_0 \in Q$ is the initial state, and $F \subset Q$ is the set of terminal states.*

The transition function, δ, tells the automaton which state to transition to depending on its current state and the next letter to be read in from a text. More precisely, $\delta(v, \sigma)$ is the state the automaton will jump to from state $v \in Q$ when the letter $\sigma \in \Sigma$ is processed. Note that $\delta(v, \varepsilon) = v$ for all $v \in Q$. Often times, however, one would like to know what state the automaton will end up at after reading a multi-letter string. For this reason, it is common to recursively extend the domain of the transition function to $\delta : Q \times \Sigma^* \to Q$ as follows:

$$\delta(q, s) := \delta(\delta(q, s[1]), s[2:]), \text{ if } |s| > 1.$$

One last concept that will be used with regards to DFA is the so-called *language of a state*. Broadly said, this is the set of strings that make the automaton transition from that state to a terminal state. In what follows, $\mathcal{L} : Q \to 2^{\Sigma^*}$ denotes the corresponding language operator i.e. for each state $q \in Q$, $\mathcal{L}(q) := \{s \in \Sigma^* : \delta(q, s) \in F\}$. Since the automaton always starts at the initial state q_0, $\mathcal{L}(q_0)$ is by definition the (regular) *language recognized* by the automaton.

1.3.1 A Toy Example

To see an example of a DFA, we consider the application in which we have a strand of DNA and wish to identify certain biological motifs within it. In this case, the reference alphabet is $\Sigma = \{A, C, G, T\}$, since these are the standard DNA bases.

Suppose that the biological motifs that we wish to look for are described by the language $L = \{AATAA, ATG, TG\}$. The DFA recognizing $\Sigma^* L$ can be seen in Figure 1.1. In this automaton, 0 is the initial state, and states 3, 6, and 9 are terminal. It should be noted that many of the labeled edges have been omitted from this figure for clarity. For example, every state should have an edge that leads back the initial state if the letter C is encountered. This is because C does not appear anywhere within L; therefore, encountering a C in the text can be thought of as a "reset."

Table 1.1 traces through the process of reading the text $AAATAATAATATTG$ from left to right. In the end, it was found that the prefixes $AAATAA$, $AAATAATAA$, and $AAATAATAATA$-TTG are strings of Σ^*L. Note that the lengths of these strings correspond to the positions in the text where a motif in L is completed. Moreover, some of the occurrences of L in the text overlap. For example, one can see that the string $AATAATAA$ contains two substrings of $AATAA$.

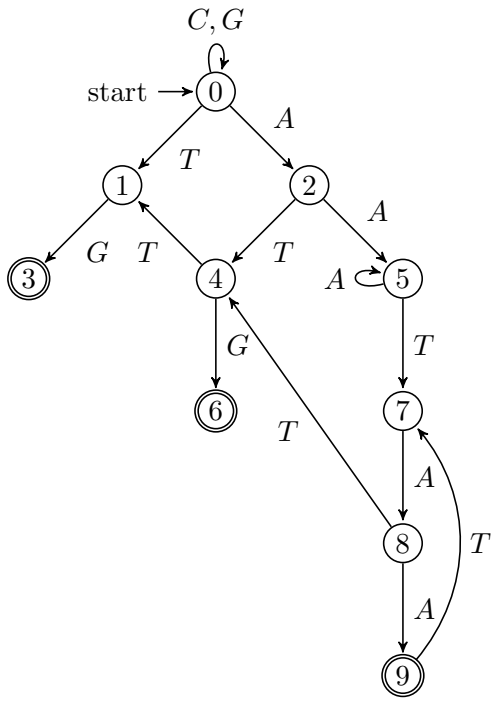

Figure 1.1: **DFA that recognizes the language** $\{A, C, G, T\}^*\{AATAA, ATG, TG\}$. For clarity, several of the labeled edges have been omitted.

Text Read In	Current State	Recognized?
ε	0	No
A	2	No
AA	5	No
AAA	5	No
AAAT	7	No
AAATA	8	No
AAATAA	9	Yes
AAATAAT	7	No
AAATAATA	8	No
AAATAATAA	9	Yes
AAATAATAAT	7	No
AAATAATAATA	8	No
AAATAATAATAT	4	No
AAATAATAATATT	1	No
AAATAATAATATTG	3	Yes

Table 1.1: **Traversal of the DFA in Figure 1.1 as the text** $AAATAATAATATTG$ **is read in from left to right.**

1.3.2 The Minimal DFA

A major focus of this paper will be centered on the minimal DFA; that is, the DFA that can successfully recognize the specified language with the least amount of states possible. Much of the terminology and ideas presented here is based on Chapter 4.4 of [5]. The reader is encouraged to refer to that chapter for a more thorough explanation, including an algorithm to minimize a pre-existing automaton.

To start, any two states p and q (regardless if they are in the same automaton) are said to be *indistinguishable* if $\mathcal{L}(p) = \mathcal{L}(q)$ (if this is not the case, these states are *distinguishable*). In other words, the two states are indistinguishable if any string that causes a transition from state p to a terminal state also causes a transition from state q to a terminal state. Note that this is not as strong as requiring that $\delta(p, s) = \delta(q, s)$, for all $s \in \Sigma^*$. Indistinguishability only requires that $\delta(p, s) \in F$ if and only if $\delta(q, s) \in F$.

In what follows, we say that a DFA $(Q, \Sigma, \delta, q_0, F)$ is *minimal* when

(1) For all states $q \in Q$, there exists a string $s \in \Sigma^*$ such that $\delta(q_0, s) = q$, i.e. q is reachable from the start state; and

(2) all states in Q are mutually distinguishable.

Theorem 1.1. *(Theorem 4.26 of [5].) A DFA that satisfies the two conditions above and recognizes a certain regular language L has the least amount of states possible among all other DFA that recognize the same language.*

Proof. The proof presented here is largely a restatement of Section 4.4.4 in [5]. It is included in the hopes that it will introduce the reader to ideas used in later proofs.

Let M be a minimal DFA that recognizes a certain regular language L. Suppose, for the sake of contradiction, that there exists such a DFA recognizing L, N, that is smaller than M.

First note that the start state of M, m_0, and the start state of N, n_0, must be indistinguishable from each other. This is because $L = \mathcal{L}(m_0) = \mathcal{L}(n_0)$. Following this, we claim that for every state m in the DFA M, there is a state n in the DFA N that is indistinguishable from m. Let s be a string such that $\delta(m_0, s) = m$. Then the language of the state m will be $\{s' \in \Sigma^* : ss' \in \mathcal{L}(m_0)\}$. To see this point more clearly, consider a string in $\mathcal{L}(m_0)$ that can be written as the concatenation ss'. Since this concatenation is in the language of m_0, $\delta(m_0, ss') = \delta(m, s')$ is a terminal state. Thus, s' is in the language of m. Furthermore it is easy to see that for any $r \in \mathcal{L}(m)$ that $r \in \{s' \in \Sigma^* : ss' \in \mathcal{L}(m_0)\}$ since $\delta(q_0, sr) = \delta(q, r) \in F$.

Let $n = \delta(n_0, s)$. By the same logic $\mathcal{L}(n) = \{s' \in \Sigma^* : ss' \in \mathcal{L}(n_0)\}$, and since $\mathcal{L}(n_0) = \mathcal{L}(m_0)$, the two states are indistinguishable. Therefore, every state m in DFA M has a corresponding, indistinguishable state n in DFA N.

Recall that we assumed that N has less states than M. Using the pigeonhole principle, there must be different states m_1 and m_2 in M that are both indistinguishable by the same state n in N. Therefore, $\mathcal{L}(m_1) = \mathcal{L}(n) = \mathcal{L}(m_2)$. However, this breaks the assumption that all states in M are mutually distinguishable, and thus, such a DFA N cannot exist. □

Note from the proof that when N is also minimal, the previous argument constructs a bijective mapping between the states of M and the states of N. Thus, up to a relabeling of states, a minimal automata that recognizes a certain regular language is unique.

1.4 Thesis Problem Statement

One of the greatest obstacles faced when using DFA is the possibility of an exponential explosion in size. In particular, for a given regular language, the set of states associated with a DFA that recognizes it may grow to be too large to store in memory. As a result, it becomes essential that the DFA that one constructs is minimal so as to use memory as efficiently as possible. To demonstrate how memory consumption can be an issue, we next introduce the notion of generalized strings, as defined by Marschall in [6].

1.4.1 Generalized Strings

A *generalized string* is a language, G, with a particular structure. Consider the case in which we have n non-empty subsets of Σ. We will denote these as $G[1], G[2], \ldots, G[n] \subset \Sigma$, and refer to them as *generalized characters* (or just characters if it is clear by context). The language G is simply the concatenation of these generalized characters, i.e. $G \coloneqq G[1]G[2]\ldots G[n]$. By definition the length of the generalized string, $|G|$, is n. Note this is different from the cardinality of G, namely $\prod_{i=1}^{n} |G[i]|$. Because of the latter, the number of strings in a generalized string typically

grows exponentially fast with its length.

For an example, consider the DNA alphabet $\Sigma = \{A, C, G, T\}$ and the generalized string $G = \{A, G\}\{A, C, G, T\}\{T\}$; in particular, $|G| = 3$. This generalized string contains $2 \cdot 4 \cdot 1 = 8$ strings. These are AAT, ACT, AGT, ATT, GAT, GCT, GGT, and GTT. Note that every string within the language has the same length, a property of all generalized strings.

We now demonstrate how generalized strings can cause the number of states to grow at an uncontrollable rate. If one wishes to take a naive approach, a DFA recognizing $\Sigma^* G$ can be formed by first constructing the prefix tree corresponding to G. To construct depth i of the aforementioned tree, one simply adds a child corresponding to each letter in $G[i]$ to every state in depth $(i - 1)$. This tree by itself is not a completed DFA, however. Additional edges need to be added to ensure that the transition function is defined for every state-letter pair.

Figure 1.2 shows an example prefix tree in which $G = \{A, G\}\{C, T\}$. For this case the number of total states in the prefix tree is $(2^3 - 1)$. More broadly speaking, if there are n generalized characters, all with the same size j, then the total number of states in the prefix tree is $j^{n+1} - 1$. Therefore increasing the generalized string even by one character can potentially be very costly in terms of memory.

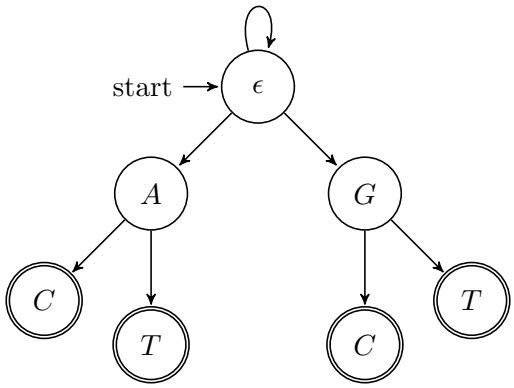

Figure 1.2: **DFA recognizing $\Sigma^* G$, with $G = \{A, G\}\{C, T\}$.** Many of the edges have been ommitted for clarity.

The reader should note that this is a naive way to construct the automaton. There are many states within the DFA that contain redundant information, and removing them would result

in significant improvements to the memory consumption. That being said, the above example demonstrates why it is essential for one to be mindful of the DFA construction and be efficient as possible with memory.

1.4.2 Thesis Overview

Motivated by the exponential explosion in state space that generalized strings can cause, throughout this thesis we will consider how to minimize the number of states needed and analyze this best case scenario. As such, we propose a new algorithm in the following chapter that constructs the minimal DFA recognizing $\Sigma^* G$.

While there are algorithms that compress pre-existing DFA [4, 7, 8], it may be the case that there is only enough computer memory to fit the minimal DFA. Thus this process of finding an arbitrary DFA and then compressing it to the minimal DFA becomes infeasible. To address this problem, Daciuk et al. created an algorithm that will form the minimal DFA directly by iteratively expanding the language of sought after words [3]; however, this algorithm is not necessarily optimized for generalized strings.

More recently, Marschall has presented an algorithm to create the minimal DFA under the context of generalized strings [6]. The algorithm constructs a non-deterministic finite automaton (NFA) in such a way so that when the so-called Subset Construction algorithm is performed on the NFA, the result is the minimal DFA. With this approach to the problem, one must first make an auxiliary structure (the NFA); thus, we seek to find an algorithm that builds the minimal DFA directly and efficiently under the context of generalized strings.

Besides eliminating the need for buffer memory, constructing the minimal DFA directly provides an easier way to do a stochastic analysis on its number of states. This is especially impactful considering that the minimal DFA is unique, and therefore its analysis is ambivalent to the algorithm chosen for construction. Therefore, in the remaining chapters, it is assumed that G is drawn from some probability distribution, allowing for a stochastic analysis of the number of states in its associated minimal DFA. In particular, a proof is given on an upper bound that is true with

probability 1 as the size of the generalized string increases. Along side this, the claim that the expected number of states in the minimal DFA grows linearly with respect to size is presented later in the thesis.

Chapter 2

A Novel DFA Construction Algorithm for Generalized Strings

In this chapter we explore a new algorithm that allows one to iteratively construct the minimal DFA for generalized strings. Since the algorithm relies heavily on the so-called *Aho-Corasick* automaton, we will first introduce the ideas behind said automaton. Following this, a statement of the algorithm will be given along with an example of its use. We will conclude this chapter by proving the correctness and minimality of the DFA produced by the algorithm.

2.1 The Aho-Corasick Automaton

Before we proceed to the statement of the algorithm, we introduce the so-called *Aho-Corasisck automaton* (abbreviated ACA) [1]. Given a *finite* language L, this is a DFA constructed to recognize the language $\Sigma^* L$, which encodes the transition function via two other auxiliary functions: the *goto function* and the *failure function*. In a very broad sense, the goto function is used when the automaton is getting closer to recognizing a string in $\Sigma^* L$, and the failure function is used otherwise.

The structure of the ACA mirrors the prefix tree associated with the strings in the finite language L. Its initial state q_0 is the root of the tree. Throughout this manuscript, we will denote this standard ACA as $M_{ACA} = (Q, \Sigma, \delta, q_0, F)$. An example of the standard ACA can be seen in Figure 2.1. Since we are considering the ACA as a tree, we can introduce the notion of depth. For each state $q \in Q$, let the depth of the state, $depth(q)$, be the length of the unique path leading from q_0 to q. As such, $depth(q_0) = 0$.

Similar to the transition function, the goto function, g, takes a state-letter pair and returns a state. The major difference between the two, however, is that the goto function need not be defined for every possible state-letter input. In fact, the values for which g is defined correspond to the edges of the prefix tree associated with L. Therefore, for $q \in Q$ and $\sigma \in \Sigma$, $depth(g(q, \sigma)) = depth(q) + 1$ whenever $g(q, \sigma)$ is defined. Since use of the goto function corresponds to progression deeper into the tree, its use can be thought of as getting closer to recognizing a string in $\Sigma^* L$. The one possible exception to this is at the initial state q_0. For any letter σ that does not have an edge going from the initial state to a state in depth 1, we define $g(q_0, \sigma) \coloneqq q_0$. Thus, the initial state is the only state in which one can guarantee that the goto function will be defined for all letters in the alphabet. Lastly, we can extend the goto function to take strings for input in the same way that it was done for transition functions. That is, the goto function is applied iteratively for each letter in the string passed in; however, if at any point during this process the goto function is not defined for some state-letter pair, then the extension is not defined for the state-string input. Following this, we refer to this extension when speaking of the goto function, g.

For the cases in which the goto function is not defined, the failure function $f : Q \setminus \{q_0\} \to Q$ is used instead. (Note that this function is not defined for the initial state.) In cases where no forward progress can be made in the tree, the failure function provides a way to jump back to a previous state in the tree instead of restarting from the start state.

To fix ideas, consider the language $L = \{ABCC, BCDA\}$ and assume that the letters ABC have been read in; in particular, the string $ABCC$ has nearly been recognized. If the letter D was read in next, however, all progress in recognizing this string would be lost, and instead only an additional A would be needed for the string $BCDA$ to be recognized. The failure function is responsible for keeping track of this type of information. By mapping states to those that occur previously in the automaton, it implicitly matches the longest possible suffix of what has been read in to a prefix in L. For instance, where $q \coloneqq \delta(q_0, ABC)$ and $p \coloneqq \delta(q_0, BC)$, $f(q) = p$ because BC is the longest suffix of ABC that is simultaneously a prefix of some word in L.

More generally, consider a state $q \in Q \setminus \{q_0\}$ and the corresponding string $s \in \Sigma^*$ such

that $g(q_0, s) = q$. Now consider any index $2 \leq i \leq (|s| + 1)$ such that $g(q_0, s[i:])$ is well-defined. Note that $s[i:]$ is a strict suffix of s that is also a prefix of a string in L. Furthermore, when i^* is the smallest of such indices, $s[i^*:]$ is the largest suffix of s that matches a prefix. We define $f(q) \coloneqq g(q_0, s[i^*:])$, and will often refer to $f(q)$ as the "failure" of q. Furthermore, we say that there is a "failure edge" going from q to $f(q)$. Note that since $s[i^*:]$ is a strict suffix of s, the failure function always maps to states of lesser depth. This is in direct contrast with the goto function, which leads to states deeper in the tree (except for possibly at the initial state).

In conjunction, the goto and failure functions can be used to reconstruct the transition function, as follows:

$$\delta(q, \sigma) = \begin{cases} g(q, \sigma) & , \text{ if } g(q, \sigma) \text{ is defined;} \\ \delta(f(q), \sigma) & , \text{ otherwise.} \end{cases}$$

In other words, the goto function is used, if defined, in order to progress deeper into the tree. If this is not the case, the failure function is used to go backwards in the tree, and the goto function is checked to see if it is defined at the failure. This process can potentially be repeated until we reach the initial state q_0, for which the goto function is defined for all letters.

We note that the construction of the goto and failure functions is relatively straightforward. Since the goto function aligns with the edges of the prefix tree (except for possibly the initial state), it is found implicitly upon constructing the tree. On the other hand, the failure function for a state q can be found by inspecting its *parent*, p. Note that because the automaton is derived from a prefix tree, each state has exactly one parent (the one exception is the initial state, which has none). The parent of q is the state in the previous depth that is connected to q by an edge described by the goto function (i.e. a "goto edge"). The algorithm for finding $f(q)$ is given by the Algorithm 1 (this algorithm was first presented as Algorithm 3 in [1]).

This algorithm computes the failure for a state, q, evaluating the failure function recursively until encountering a state where the goto function is well-defined. When such a state is reached, the goto function is used and the resulting state is marked as the failure for q. Note that the this process will always find $f(q)$ because the initial state has the goto function defined for all letters. In

Algorithm 1: Finding the Failure Transition

INPUT: The state $q \in Q \setminus \{q^*\}$ for which we want to calculate $f(q)$
Let $p \in Q$ be the parent state of q and $\sigma \in \Sigma$ to be the letter such that $g(p, \sigma) = q$
while $f(q)$ has not been found **do**
  Set $p = f(p)$
  if $g(p, \sigma)$ exists **then**
    Set $f(q) = g(p, \sigma)$
    Return
  end if
end while

this way, the algorithm builds upon the suffixes matching with prefixes that have been previously determined.

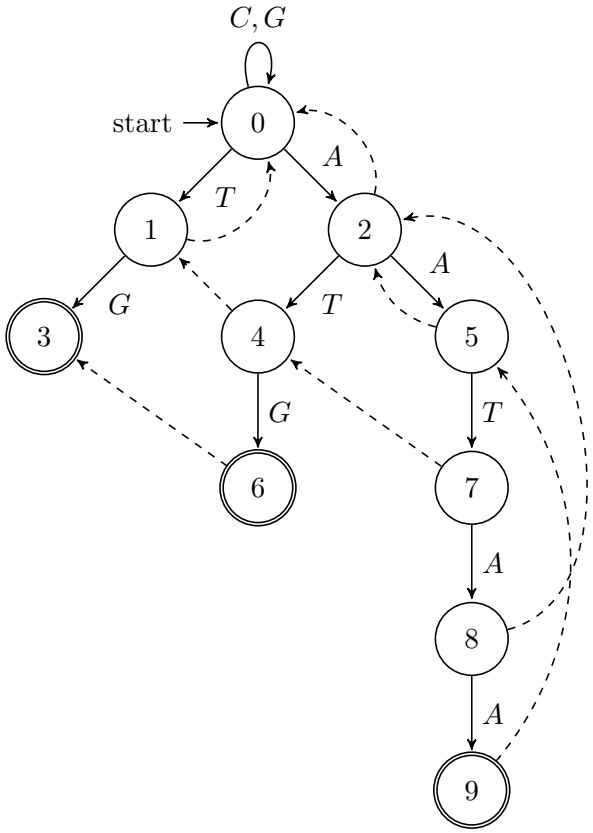

Figure 2.1: **The standard ACA recognizing the language** $\{A, C, G, T\}^*\{AATAA, ATG, TG\}$. Terminal states are denoted by double circles. Solid edges represent the goto function, while dashed edges represent the failure function.

As an example, consider again the DFA recognizing language $\Sigma^* L$, with $\Sigma = \{A, C, G, T\}$

and $L = \{AATAA, ATG, TG\}$ (see Figure 1.1). This time, however, we encode the DFA as an ACA as shown in Figure 2.1. Note that with this representation the graph contains less edges overall. In general, encoding the transition function via the goto and failure function is more memory efficient. The drawback to this, however, is that multiple evaluations of the failure and goto functions may be needed to make just one transition. For instance, if the current state of the automaton is 7 and, the letter C is read in, the failure function is evaluated three times (transitioning from states 7, to 4, to 1, and to 0) and the goto function is evaluated once (transitioning from state 0 to 0). In particular, four evaluations are needed to determine that $\delta(7, C) = 0$.

2.2 Statement of New Algorithm

We are now ready to state a new algorithm that constructs the minimal DFA recognizing $\Sigma^* G$, where G is a finite generalized string. Since the output of the algorithm is an automaton encoded via goto and failure functions, we denote it as M^*_{ACA}. This algorithm is a specialization of an algorithm presented by AitMous, Bassino, and Nicaud [2] for generalized strings.

The new Algorithm 2 exploits that generalized strings lead to very structured prefix trees, and that the language of a state can be partitioned into two sets of strings: those which cause a transition to a terminal state using only goto edges, and those that use at least one failure edge.

2.3 A Toy Example

To bolster the ideas of Algorithm 2, we show how to construct the minimal DFA recognizing $\Sigma^* G$ for the alphabet $\Sigma = \{A, B, C, D\}$ and the generalized string $G = \{A, C, D\}\{B, C\}\{A, D\}$.

The algorithm first creates the initial state q_0, expands the nodes that will be in depth 1 using $G[1]$, and constructs the goto function at q_0 (see Figure 2.2(a)). After this initial set up, it finds the failures for each state in depth 1 (see Figure 2.2(b)). But, because all of the states in depth 1 have q_0 as their failure, the algorithm merges all these states into a single one represented by $\{A, C, D\}$ (see Figure 2.2(c)). Note that no matter the generalized string under consideration, the states in depth 1 will always merge together into a single state. This is because the failure

Algorithm 2: Minimal DFA Construction for Genearlized Strings

```
Initialize empty automaton M*_ACA.
Add q_0 to Q
for each σ ∈ Σ do
  if σ ∈ G[1] then
    Add a new state q to Q
    Set g(q_0, σ) = q
  else
    Set g(q_0, σ) = q_0
  end if
end for
for d ∈ {1, 2, ..., |G|} do
  Calculate f(q) for each q ∈ {q ∈ Q : depth(q) = d} using Algorithm 1.
  Merge states in depth d with identical failures.
  Update g for states in depth (d − 1) to reflect merging.
  for each q ∈ {q ∈ Q : depth(q) = d} do
    if d < |G| then
      for each σ ∈ G[d + 1] do
        Add a new state r to Q
        Set g(q, σ) = r
      end for
    else
      Add q to F
    end if
  end for
end for
return M*_ACA
```

function represents the matching of longest proper suffixes with prefixes in the generalized string. Thus, the failures for states in depth 1 must necessarily be q_0 since the only proper suffix is ε.

Next, states in depth 2 are expanded using $G[2] = \{B, C\}$ and the respective failures are found. No merging can be done in this depth because the two failures are unique (see Figure 2.3).

Lastly, using $G[3] = \{A, D\}$, two states are added in depth 3 for each state in depth 2 (see Figure 2.4(a)). The states in depth 3 are then merged into a single state because each of their failures are the same state. Since this is the last depth of the automaton, the states are marked as accepting states (see Figure 2.4(b)). Due to theorems 2.1 and 2.2 in the Section 2.5, this is the minimal DFA that recognizes the language $\Sigma^* L$. Note that this resulting automaton is significantly smaller than the standard ACA presented in Figure 2.1. This is despite the fact that the two the languages G and $\{AATAA, ATG, TG\}$ are relatively comparable (in fact G has a

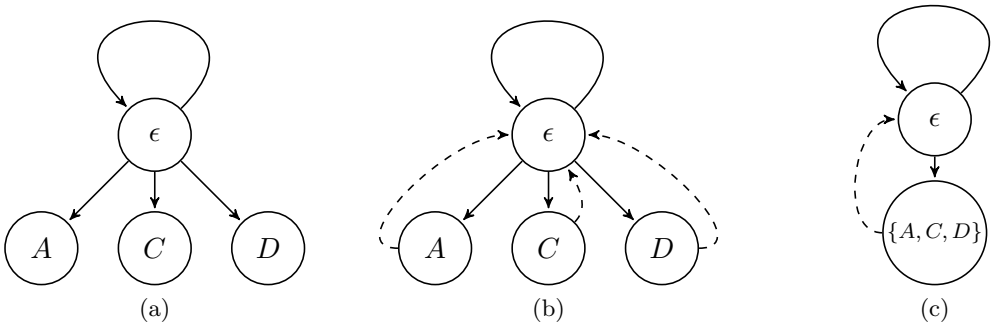

Figure 2.2: **Steps to construct the first depth in the minimal automaton recognizing Σ^*G, with $\Sigma = \{A, B, C, D\}$ and $G = \{A, C, D\}\{B, C\}\{A, D\}$.** As usual, the dashed edges represent the failure function, while the solid edges represent the goto function. (a): The automaton right after the states in depth 1 have been added. (b): The automaton after the failure function has been updated for the first depth. (c): The automaton after the states on the first depth have been merged.

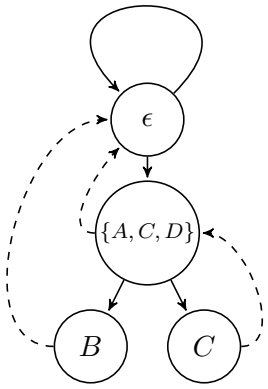

Figure 2.3: **The automaton after the failure function was updated for the second depth.**

higher cardinality).

2.4 Well Definedness

In order to further analyze the result of Algorithm 2, we must show the following.

Lemma 2.0.1. *The output of Algorithm 2 is a valid DFA.*

Proof. To show that this is true, all states in the resulting automaton should have a transition corresponding to each letter in Σ. Note that this must be the case for the initial state since it remains unchanged from the original ACA. Furthermore, all other states are either left over or

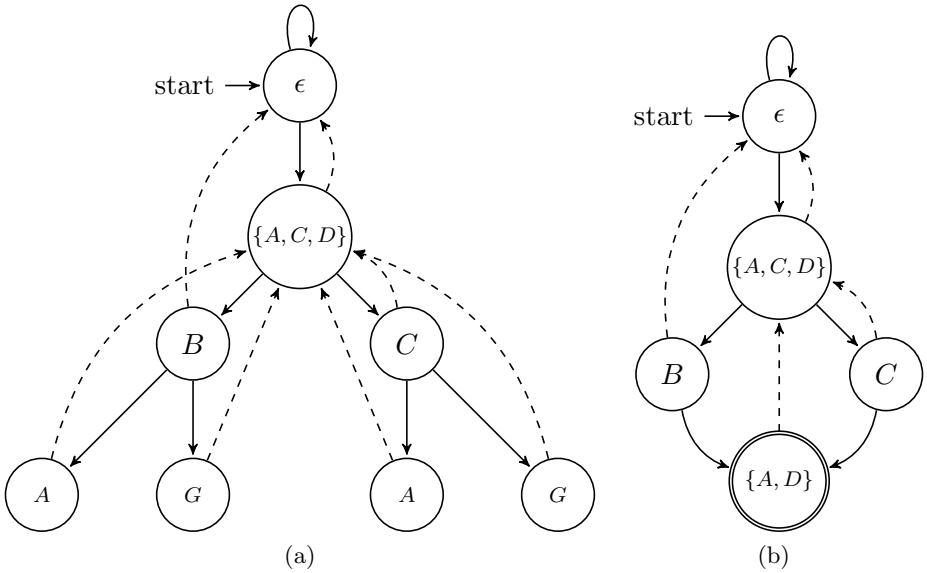

Figure 2.4: **The final steps for constructing the minimal automaton recognizing $\Sigma^* G$, with $\Sigma = \{A, B, C, D\}$ and $G = \{A, C, D\}\{B, C\}\{A, D\}$.** (a): The automaton after the third depth has been expanded and the failures have been calculated. (b): The final, resulting automaton the Algorithm 2 produces with goto and failure functions.

merged from states in the original ACA; in particular, all states in the resulting automaton have goto and failure functions, from which a valid transition function can be made. Lastly, since the original ACA was deterministic, the output of the algorithm must be as well. □

2.5 Correctness and Minimality

From this point onwards when referring to an ACA we assume that the goto functions trace out a tree, but with one exception. For any state (other than the start state), we allow for there to be multiple parents as long as the parents are in the same depth. Furthermore, we assume that every state is reachable from the start state via goto edges. Note that the ACA produced by Algorithm 2 still fits this description.

Although the class of automata described here are not strictly trees, this description still allows us to use the notion of depth (i.e. the length of the path from start state to the given state). Another concept that can be used with these tree-like ACA is described by the following:

$$\mathcal{L}_*(q) \coloneqq \{s \in \mathcal{L}(q) : |s| = |G| - depth(q)\}.$$

In other words, $\mathcal{L}_*(q)$ is the subset of strings in $\mathcal{L}(q)$ that allows the automaton to transition directly to a terminal state without the use of any failure edges.

Lemma 2.0.2. *Let G be a generalized string, $M = (Q, \Sigma, \delta, q_0, F)$ be a tree-like ACA as previously described recognizing $\Sigma^* G$, and $q \in Q \backslash \{q_0\}$ be a state in the ACA. Then $\mathcal{L}_*(q)$ and $\mathcal{L}(f(q))$ partition $\mathcal{L}(q)$.*

Proof. To start, fix $q \in Q \backslash \{q_0\}$ and let s be a string such that $\delta(q_0, s) = q$ and $|s| = depth(q)$ (i.e. s leads directly to q without the use of any failure transitions).On top of this, let $i \in \{2, 3, \ldots, |s|+1\}$ be the starting index for the longest strict suffix of s that is a prefix in G (for $i > |s|$ the suffix is ε). By the definition of the failure transition, $\delta(q_0, s[i:]) = f(q)$. Note that the value of $f(q)$ and the index i is necessarily ambivalent to the string, s, chosen.

With this established, we begin by proving that $\mathcal{L}(f(q)) = \mathcal{L}(q) \setminus \mathcal{L}_*(q)$. As a first step, we show that $\mathcal{L}(f(q)) \subseteq \mathcal{L}(q) \setminus \mathcal{L}_*(q)$. Consider a string $x \in \mathcal{L}(f(q))$. For such a string we know that, $\delta(f(q), x) \in F$. Thus $\delta(q_0, s[i:]x) \in F$ and $s[i:]x \in \mathcal{L}(q_0)$.

Because we have constructed the automaton to recognize all suffixes that are in the generalized string, $\Sigma^* s[i:]x \subset \mathcal{L}(q^*)$. Following this logic we note that $sx \in \mathcal{L}(q_0)$, implying that $\delta(q_0, sx) \in F$ and $\delta(q, x) \in F$. This shows that $x \in \mathcal{L}(q)$; however, also note that $x \notin \mathcal{L}_*(q)$ because $depth(f(q)) < depth(q)$, and therefore $|x| > |G| - depth(q)$. Thus one can see that $x \in \mathcal{L}(q) \setminus \mathcal{L}_*(q)$ and that $\mathcal{L}(f(q)) \subseteq \mathcal{L}(q) \setminus \mathcal{L}_*(q)$.

Now we aim to show $\mathcal{L}(q) \setminus \mathcal{L}_*(q) \subseteq \mathcal{L}(f(q))$. Consider a string $x \in \mathcal{L}(q) \setminus \mathcal{L}_*(q)$. Here we know that $\delta(q, x) \in F$, implying that $\delta(q_0, sx) \in F$ and $sx \in \mathcal{L}(q_0)$. Since $x \notin \mathcal{L}_*(q)$, we know that the instance of G that was recognized is a strict suffix of sx. In particular, for some $j \in \{2, 3, \ldots, |sx|\}$ we know that $(sx)[j:] \in \mathcal{L}_*(q_0)$ and that $\Sigma^*(sx)[j:] \subset \mathcal{L}(q^*)$. Additionally note that $i \leq j$ because $s[i:]$ is the largest suffix that is a prefix for a string in G by definition of the failure function. Thus $(sx)[j:]$ is a suffix of $s[i:]x$. Using the fact that $\Sigma^*(sx)[j:] \subset \mathcal{L}(q_0)$, one can see $s[i:]x \in \mathcal{L}(q_0)$ and $x \in \mathcal{L}(f(q))$. Therefore we see that $\mathcal{L}(q) \setminus \mathcal{L}_*(q) \subseteq \mathcal{L}(f(q))$, and moreover that,

$$\mathcal{L}(q) \setminus \mathcal{L}_*(q) = \mathcal{L}(f(q))$$

Finally, note that $\mathcal{L}_*(q) \subset \mathcal{L}(q)$, implying that $\mathcal{L}_*(q)$ and $\mathcal{L}(f(q))$ form a partition of $\mathcal{L}(q)$. □

We are now ready to prove that the algorithm constructs an ACA correctly identifying strings in $\Sigma^* G$. Subsequently, it will be proved that this ACA is indeed the minimal DFA to do so.

Theorem 2.1. *(Correctness.) Given the generalized string G, the automaton constructed with Algorithm 2 recognizes the language $\Sigma^* G$.*

Proof. For simplicity, we imagine the algorithm acting on the standard ACA, M_{ACA}, rather than constructing the resulting automaton directly. Let a and b be two states in M_{ACA} with the same depth such that $f(a) = f(b)$. We show that after the merging operation taken in Algorithm 2 is applied to M_{ACA}, the resulting automaton will recognize the same language.

To start, we note that $\mathcal{L}_*(a) = \mathcal{L}_*(b)$ by the nature of generalized strings. Furthermore, $\mathcal{L}(f(a)) = \mathcal{L}(f(b))$ since $f(a) = f(b)$. Therefore, by Lemma 2.0.2 we see that $\mathcal{L}(a) = \mathcal{L}(b)$, implying that these two states can be merged without consequence.

However, in the true algorithm we are constructing the automaton directly, not from a preformed M_{ACA}. As such, we also merge together pairs of children between the states a and b. In particular, given an appropriate string s', states $a' = g(a, s')$ and $b' = g(b, s')$ will be merged. Note that where a' exists a corresponding b' always exists because of the structure of a generalized string.

To show why this action preserves the language recognized by the automaton, note that for an arbitrary state q and its child $q' = g(q, s')$ that,

$$\mathcal{L}(q') = \{s[|s'| + 1 :] : s \in \mathcal{L}(q) \text{ and } s \text{ has prefix } s'\}$$

In other words, the subset of strings in $\mathcal{L}(q)$ that have the prefix s' contains strings that lead to an accepting state from q but also pass through the state q'. Therefore, removing the prefix s' from each of these strings results in the set of all possible strings that lead to an accepting state from q'.

By this property and from the fact that $\mathcal{L}(a) = \mathcal{L}(b)$, it becomes clear that $\mathcal{L}(a') = \mathcal{L}(b')$, indicating that the two children can be merged together. Therefore since all pairs of states that were merged have the same language, we know that the new, reduced automaton will recognize the same language.

Algorithm 2 simply performs a finite number of these merging operations, and we have shown that after each operation the language recognized by the automaton is conserved. We can therefore conclude that the constructed automaton recognizes the same language as M_{ACA}; that is, it recognizes the language $\Sigma^* G$. □

Theorem 2.2. *(Minimality.) Given a DFA M^*_{ACA} formed by Algorithm 2 to recognize the language $\Sigma^* G$, M^*_{ACA} is the minimal DFA that recognizes this language.*

Proof. Recall that M^*_{ACA} is minimal if we can reach all states from the initial state and if no two states have the same language. The former statement is trivially true since all new states are appended to paths rooted at the initial state. The latter statement we will show by leveraging the goto and failure functions.

Note that for a state $q \in Q$, $\mathcal{L}(q)$ only contains words of length greater than or equal to $|G| - \mathit{depth}(q)$. In particular, if $a, b \in Q$ have different depths then $\mathcal{L}(a) \neq \mathcal{L}(b)$ i.e. a and b are distinguishable. Therefore, to prove the theorem it suffices to show that different states at the same depth are non-equivalent. To do this we use strong induction. The base case is trivial because q_0 is the only state with depth 0.

Thus, assume that all states with depth at most $k \geq 0$ are distinguishable, and suppose there are unique states $a, b \in Q$ such that $\mathit{depth}(a) = \mathit{depth}(b) = (k+1)$ (if there is only one state, the state is trivially distinguishable). As we have observed before, it must be true that $\mathcal{L}_*(a) = \mathcal{L}_*(b)$ because of the structure of generalized strings. Therefore, one must compare strings in the languages with greater length to find a difference.

Note that $f(a) \neq f(b)$ because if the two were equal, a and b would have been merged during construction. Furthermore, by the inductive hypothesis it was assumed that $\mathcal{L}(f(a)) \neq \mathcal{L}(f(b))$.

Using Lemma 2.0.2 we can therefore say that $\mathcal{L}(a) \neq \mathcal{L}(b)$ for any such valid a and b. Thus we see by induction that all states within a depth are also mutually distinguishable. From this we conclude that the automaton M^*_{ACA} is minimal. $\square$

Chapter 3

Stochastic Analysis of Memory Complexity

In this chapter, we get a sense of how much memory is consumed by the minimal DFA constructed for a generalized string. Although the approach for this will be through inspection of Algorithm 2, recall that the minimal automaton is unique. Thus our analysis applies to all minimal DFA constructed for generalized strings, regardless of the algorithm used. Throughout the chapter, the goal is to find an upper bound on the number of states in any such minimal DFA. Since the initial deterministic approach leads to a loose upper bound, we turn to random generalized strings. Our theoretical results are demonstrated by synthetic data.

3.1 A Deterministic Upper Bound

To start, we give an upper bound for the size of the state space. While a similar bound was noted in [6] by Marschall, here we are able to use Algorithm 2 to find that this previous upper bound can be reduced by a factor of two.

Proposition 3.0.1. *If G is a generalized string and $M^*_{ACA} = (Q, \Sigma, \delta, q_0, F)$ is the minimal DFA recognizing $\Sigma^* G$ then $|Q| \leq 2^{|G|}$.*

Proof. Let $n := |G| \geq 1$. The proof is via induction on n. Recall that depth 1 of the DFA constructed by Algorithm 2 always has a single state. This is because every failure stemming from a state at this depth must necessarily go to the root state, and thus all states will be merged into one. Therefore, when $n = 1$, $|Q| = 2 \leq 2^1$.

Next, assume that the proposition holds for certain $n \geq 1$. Since Algorithm 2 merges states with identical failures, another way one can think of the number of states in a given depth is by considering the amount of unique failure transitions at that depth. Note that this number is limited by the number of states that come before the depth in question. Consider now a generalized string of length $(n+1)$ and let M^*_{ACA} be the associated DFA constructed by Algorithm 2. By the previous logic, the number of states in depth $(n+1)$ can be at most the number of states in depths 0 through n. In particular, if $\hat{Q} \subset Q$ is the set of states of depth n or less in M^*_{ACA} then $|Q| \leq |\hat{Q}| + |\hat{Q}|$. Then, by the inductive hypothesis, $|Q| \leq 2|\hat{Q}| \leq 2 \cdot 2^n = 2^{n+1}$. This shows the proposition. □

While the upper bound in the proposition grows exponentially fast, it still provides a significant improvement over the $\mathcal{O}(|\Sigma|^{|G|})$ worst case growth of the prefix tree associated with $|G|$. That being said, as seen in Figure 3.1, this bound is still loose for even short generalized strings. In retrospective, this is not surprising. Indeed, as one can see from the proof of the proposition, achieving this bound would imply that within each depth there are failure edges leading back to every possible state, which seems very unlikely.

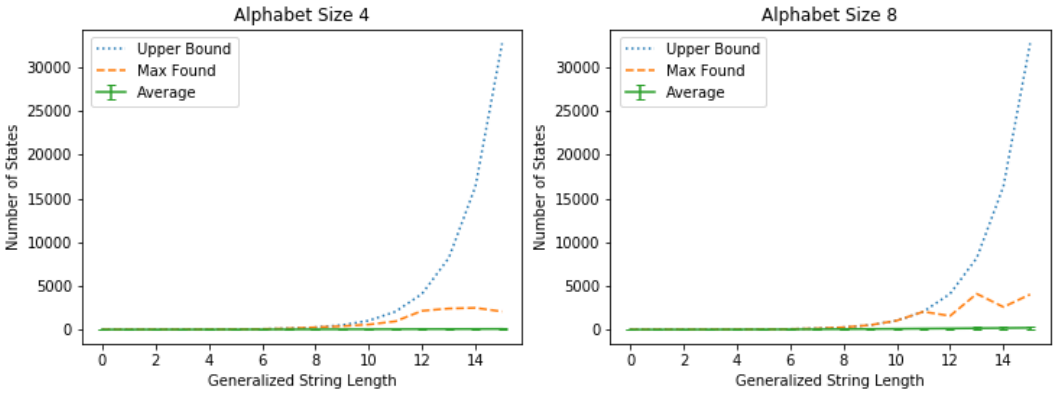

Figure 3.1: **Comparison of deterministic upper bound in Proposition 3.0.1, average of synthetic data, and the maximum seen in synthetic data with regards to number of total states.** For these plots, alphabet sizes of 4 (left) and 8 (right) were used. For each generalized string length, 10K generalized strings were drawn uniformly at random and the corresponding minimal DFA were constructed.

3.2 Linking Number of States to Matching Suffixes to Prefixes

In order to create a more practical bound, it behooves one to consider the nature of failure functions with regard to generalized strings. One approach to doing this is to view the problem through the lens of the failure function connecting longest suffixes to prefixes.

To fix ideas, consider the alphabet $\Sigma = \{A, B, C, D\}$ and $G = \{A, C\}\{C, D\}\{A, B\}\{B, D\}$. Notice that $G[1] \cap G[3] = \{A\} \neq \emptyset$ and $G[2] \cap G[4] = \{D\} \neq \emptyset$, leading to the conclusion that there is at least one string in G where the last two letters match with the first two letters of another string in G. For example, the last two letters of $ACAD$ match with the first two letters of $ADBB$ (other examples are possible). Thus, the longest suffix matching a prefix in G must be at least of length two. This insight informs what kind of failures can appear in the last depth of the corresponding minimal DFA.

In order to analyze the behavior of the failure function in other depths of the automaton, the same process can be performed but with a prefix of the generalized string G. That is, to analyze the failure function in depth i of the automaton, one can find the longest suffixes that match with prefixes for $G[: i]$. Doing this is validated by the following lemma.

Lemma 3.0.1. *Consider an integer $n \geq 0$ and generalized strings G_1 and G_2 such that $|G_1| \geq n$ and $|G_2| \geq n$. Let M_1 and M_2 be the minimal automata associated with G_1 and G_2, respectively. If $G_1[: n] = G_2[: n]$ then M_1 and M_2 have identical states and failure functions up to depth n.*

Proof. In what follows, we assume without any loss of generality that M_1 and M_2 are constructed via Algorithm 2.

An inductive argument is used to prove the lemma. The base case in which $n = 0$ is trivial because M_1 and M_2 each have one state in depth 0 (the start state), and the failure function is not defined on the start state for either automaton.

Now assume the statement holds for some $n \geq 0$. Furthermore, suppose that $G_1[: n+1] = G_2[: n+1]$, and that M_1 and M_2 have been constructed up to depth n. Since $G_1[n+1] = G_2[n+1]$, the states and corresponding failures that are added for depth $(n+1)$ are dependent only on states

at depth n or lower. Therefore the states and corresponding failures of depth $(n+1)$ will be identical in the two automata since, by they inductive hypothesis, they are identical up to depth n. Thus, the lemma holds for all $n \geq 0$ as stated. □

To simplify the notation, in what follows we write $G[i_1 : i_2] \cap G[j_1 : j_2] \neq \emptyset$ to mean that $G[i_1] \cap G[j_1] \neq \emptyset, \ldots, G[i_2] \cap G[j_2] \neq \emptyset$. (Here, it is implicit that $1 \leq i_1 \leq i_2 \leq |G|$, $1 \leq j_1 \leq j_2 \leq |G|$, and $(i_2 - i_1) = (j_2 - j_1)$.) Note in particular that $G[i_1 : i_2] \cap G[j_1 : j_2] = \emptyset$ means that for some $0 \leq k \leq (i_2 - i_1)$, $G[i_1 + k] \cap G[j_1 + k] = \emptyset$.

Lemma 3.0.2. *Consider a generalized string, G, and its corresponding minimal DFA, $M^*_{ACA} = (Q, \Sigma, \delta, q_0, F)$. Let $1 \leq i \leq j \leq |G|$ and define $k = (j - i) + 1$. Then $G[1 : k] \cap G[i : j] \neq \emptyset$ if and only if there exists $q \in Q$ with $depth(q) = j$ such that $depth(f(q)) \geq k$.*

Proof. ($\Rightarrow$) We first assume the case in which there are k pairs of generalized characters that share at least one letter (i.e. $G[: k] \cap G[i : j] \neq \emptyset$). Much like what was seen in the previous example, there must be a string $s \in G[: j]$ such that s has a k-lengthed suffix, $s[i :]$, matching with some prefix in G. Furthermore, note that for $q \coloneqq \delta(q_0, s)$, $depth(q) = j$. Let i^* be the index such that $s[i^* :]$ is the longest suffix of s that matches with a prefix. Because $|s[i :]| \leq |s[i^* :]|$ and both suffixes match with some prefix in G,

$$depth(\delta(q_0, s[i :])) = k \leq depth(\delta(q_0, s[i^* :])) = depth(f(q)).$$

Thus depth j has at least one state that has a failure at depth k or greater.

($\Leftarrow$) We now assume that there is a $q \in Q$ with $depth(q) = j$ and $depth(f(q)) = t \geq k$. Consider a string s such that $\delta(q_0, s) = q$. It then follows that $\delta(q_0, s[t :]) = f(q)$. Based on the definition of the failure function, there exists string $r \in G[: j]$ such that $r[: t] = s[t :]$. In order for r and s to be in the language G, it must be the case that $G[: t] \cap G[j - t + 1 : j] \neq \emptyset$. Since $t \geq k$, this proves the lemma. □

The contrapositive statement of Lemma 3.0.2 will often be of more use to us. That is $G[: k] \cap G[i : j] = \emptyset$ if and only if there is no state in depth j of the automaton that has a failure

of depth k or higher.

3.3 Overlapping Generalized Strings

Let G be a generalized string of length j. Furthermore, let G_1 and G_2 be its prefix and suffix of length k, respectively. In particular, $G_1 = G[:k]$ and $G_2 = G[j-k+1:]$. It is assumed that k is large enough so that the strings share some characters. Define $t := (j-k)$, i.e. t is the number of generalized characters that G_1 does not share with G_2, and viceversa. In this section, we reveal some implicit periodicity in G when G_1 and G_2 overlap i.e. $G_1 \cap G_2 \neq \emptyset$. This periodicity is important for the probabilistic analysis of the next section.

First suppose that $2t \geq k$ i.e. that the number of shared characters between G_1 and G_2 is at most half of the length of each string. To fix ideas, consider the generalized strings in Figure 3.2. There, $j = 13$, $k = 8$, and $t = 5$. Because $G_1 \cap G_2 \neq \emptyset$, it follows that $G_1[1] \cap G_2[1] \neq \emptyset$ and $G_1[6] \cap G_2[6] \neq \emptyset$. In other words, $G[1]$ overlaps with $G[6]$, and $G[6]$ overlaps with $G[11]$. Similarly, $G[2]$ overlaps with $G[7]$, which overlaps with $G[12]$. And, $G[3]$ overlaps with $G[8]$, which overlaps with $G[13]$. Finally, because $G_1[4] \cap G_2[4] \neq \emptyset$, $G[4]$ overlaps with $G[9]$. Similarly, $G[5]$ overlaps with $G[10]$. The generalized string G therefore exposes the periodicity represented in Figure 3.2.

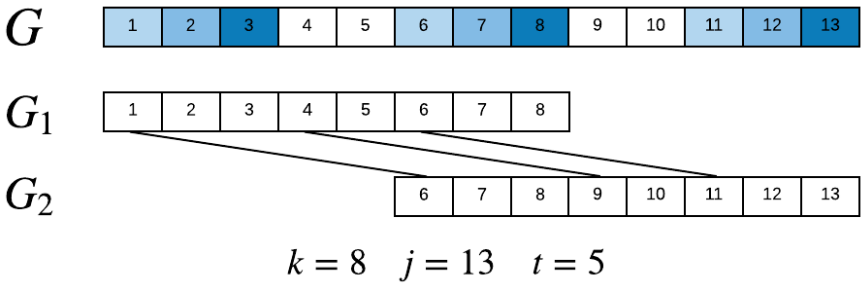

Figure 3.2: **Example of overlapping generalized strings where the number of shared characters is less than half of the string length.** The top row shows the full generalized string G, while subsequent rows show its prefix G_1 and suffix G_2. Each rectangle represents a generalized character and lines between rectangles imply the generalized characters share at least one letter (not all lines included). The coloring in G is intended to show the periodic nature of overlapping generalized substrings.

Instead if $2t \leq k$, i.e. the number of shared characters between G_1 and G_2 is at least half of the length of each string, a more complex periodic structure emerges in G. As an example, consider

the generalized strings in Figure 3.3 where $j = 14$, $k = 11$, and $t = 3$. Much like in the previous example, it follows now that $G[1]$ overlaps with $G[4]$, $G[4]$ overlaps with $G[7]$, $G[7]$ overlaps with $G[10]$, and finally $G[10]$ overlaps with $G[13]$. Note that all these pairwise overlaps occur at indices of the form $(1 + wt)$ where $0 \leq w \leq 4$. A similar conclusion applies to $G[2]$, $G[5]$, $G[8]$, $G[11]$, and $G[14]$, as well as $G[3]$, $G[6]$, $G[9]$, and $G[12]$.

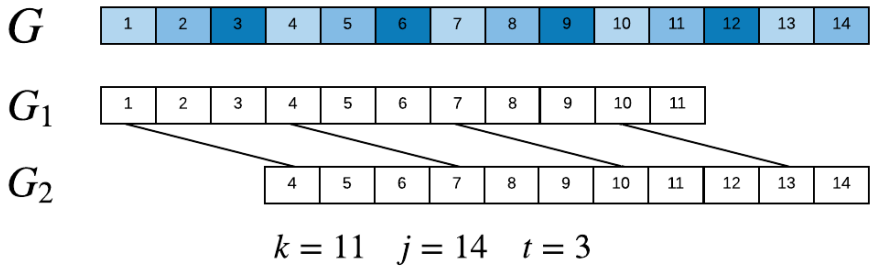

Figure 3.3: **Example of overlapping generalized strings where the number of shared characters is more than half of the string length.** In terms of the values presented in Lemma 3.0.3, $r = 2$ and $d = 3$.

These cases are presented as particular instances of the proceeding lemma.

Lemma 3.0.3. *Let G be a generalized string of length j. Suppose that there is a k-length suffix in G matching with a prefix i.e. $G[:k] \cap G[j-k+1:] \neq \emptyset$, and that k is large enough so that the $G[:k]$ and $G[j-k+1:]$ share at least one generalized character. Define $t \coloneqq (j-k)$.*

(1) If $2t \geq k$ then $G[i] \cap G[t+i] \neq \emptyset$ and $G[t+i] \cap G[2t+i] \neq \emptyset$ for $1 \leq i \leq (k-t)$, and $G[i] \cap G[t+i] \neq \emptyset$ for $(k-t) < i \leq t$.

*(2) Define $d \coloneqq \lfloor k/t \rfloor$ and $r \coloneqq k$ (**mod** t). If $2t < k$ then $G[i+wt] \cap G[i+(w+1)t] \neq \emptyset$ for $1 \leq i \leq t$ and $0 \leq w < d$, and $G[dt+i] \cap G[(d+1)t+i] \neq \emptyset$ for $(k-r+1) \leq i \leq k$.*

Proof. For simplicity define $G_1 \coloneqq G[:k]$ and $G_2 \coloneqq G[j-k+1:]$. We first consider the case in which $2t \geq k$. Because $G_1 \cap G_2 \neq \emptyset$, we must have that $G_1[i] \cap G_2[i] \neq \emptyset$ and that $G_1[t+i] \cap G_2[t+i] \neq \emptyset$ for $1 \leq i \leq (k-t)$. But note that $G_1[t+i] = G_2[i]$ because G_1 and G_2 share generalized characters. Therefore $G_1[i] \cap G_2[i] \neq \emptyset$ and $G_2[i] \cap G_2[i+t] \neq \emptyset$ for $1 \leq i \leq (k-t)$. In terms of the original generalized string G, this is equivalent to having $G[i] \cap G[t+i] \neq \emptyset$ and $G[t+i] \cap G[2t+i] \neq \emptyset$. This

gives information about all but $2(2t-k)$ generalized characters in G. Since none of these characters are shared between G_1 and G_2, we can only guarantee that $G_1[i] \cap G_2[i] \neq \emptyset$ i.e. $G[i] \cap G[t+i] \neq \emptyset$ for $(k-t) < i \leq t$. This proves the first case in the lemma.

Assume now that $2t < k$, and observe that $k = td + r$. Because $G_1 \cap G_2 \neq \emptyset$, it follows that $G_1[i + wt] \cap G_2[i + wt] \neq \emptyset$ for $1 \leq i \leq t$ and $0 \leq w < d$. But, much like the previous case, we can leverage the fact that $G_2[: t]$ consists of generalized characters shared between G_1 and G_2 to assert that $G_1[i + wt] = G_2[i + (w-1)t]$ when $w > 1$. As a result, $G[i + wt] \cap G[i + (w+1)t] \neq \emptyset$ for $1 \leq i \leq t$ and $0 \leq w < d$. This accounts for $t(d+1)$ generalized characters in G, leaving r characters at its right end which have not yet been accounted for. But, again because $G_1 \cap G_2 \neq \emptyset$, we have for $(k - r + 1) \leq i \leq k$ that $G_1[i] \cap G_2[i] \neq \emptyset$ i.e. $G[dt + i] \cap G[(d+1)t + i] \neq \emptyset$. This completes the proof of the lemma. □

3.4 A Probabilistic Perspective

In this section we take a probabilistic perspective to investigate what can be said about the size of the minimal DFA with high probability. For this consider a random generalized string G of length n such that $G[i]$, with $1 \leq i \leq n$, are i.i.d. generalized characters which satisfy that $\mathbb{P}(G[i] = \emptyset) = 0$.

Define c_2 to be the probability that any two random generalized characters in G share at least one letter. That is,

$$c_2 \coloneqq \mathbb{P}(G[i] \cap G[j] \neq \emptyset), \quad \text{for different } i \text{ and } j.$$

Note that the definition of c_2 does not depend on i nor j as long as $i \neq j$. Another important concept that warrants attention is the case in which some generalized characters $G[i]$ and $G[j]$ share at least one letter and $G[j]$ and $G[k]$, with $k \neq i$ and $k \neq j$, share at least one letter (note this does not imply that $G[i]$ and $G[k]$ share a letter). We denote the probability of this case happening as c_3. Namely:

$$c_3 \coloneqq \mathbb{P}(G[i] \cap G[j] \neq \emptyset, G[j] \cap G[k] \neq \emptyset), \quad \text{for distinct } i, j, \text{ and } k.$$

An important fact to keep in mind is that $c_3 \leq c_2$ because the event of two pairs sharing a letter is more restrictive than just one pair sharing a letter.

3.4.1 The Uniform Case

A common scenario that one may be interested in is where the distribution over non-empty generalized characters is uniform. In this case, the probability that a generalized character takes on any non-empty subset of the alphabet is $\frac{1}{2^a-1}$, where $a \coloneqq |\Sigma|$ is the size of the reference alphabet.

Proposition 3.0.2. *In the case where generalized characters are drawn uniformly at random among non-empty subsets of* Σ, $c_2 = \frac{4^a-3^a}{(2^a-1)^2}$ *and* $c_3 = \frac{8^a-(2)6^a+5^a}{(2^a-1)^3}$. *In particular,* $c_2^2 \leq c_3$ *with equality if and only if* $a = 1$.

Proof. The derivation of c_2 is relatively straightforward if we approach the problem by considering the complementary event, i.e. $c_2 = 1 - \mathbb{P}(G[i] \cap G[j] = \emptyset)$. A counting argument can be used to calculate the latter probability. Indeed, for two generalized characters to share no letters, each letter can either appear in one of the sets but not the other or in neither of the sets. There are 3^a different ways to do this. However, with this reasoning we are over counting since we cannot allow either of the generalized characters to be empty. Therefore we must subtract $(2 \cdot 2^a - 1)$ i.e. $(2^{a+1} - 1)$ to obtain

$$c_2 = 1 - \frac{3^a - 2^{a+1} + 1}{(2^a - 1)^2} = \frac{(2^a - 1)^2 - 3^a + 2^{a+1} - 1}{(2^a - 1)^2} = \frac{4^a - 3^a}{(2^a - 1)^2}.$$

Computing c_3 requires a bit more work. The main challenge is that the events $G[i] \cap G[j] \neq \emptyset$ and $G[j] \cap G[k] \neq \emptyset$ are not independent. That being said, if one knew $G[j]$, the two events would become independent. We can leverage this conditional independence and the law of total probability to obtain

$$\begin{aligned} c_3 &= \sum_{g \subset \Sigma, g \neq \emptyset} \mathbb{P}\left(G[i] \cap G[j] \neq \emptyset, G[j] \cap G[k] \neq \emptyset | G[j] = g\right) \cdot \mathbb{P}(G[j] = g) \\ &= \sum_{g \subset \Sigma, g \neq \emptyset} \mathbb{P}\left(G[i] \cap G[j] \neq \emptyset | G[j] = g\right) \cdot \mathbb{P}\left(G[j] \cap G[k] \neq \emptyset | G[j] = g\right) \cdot \mathbb{P}(G[j] = g) \\ &= \frac{1}{2^a - 1} \sum_{g \subset \Sigma, g \neq \emptyset} \mathbb{P}\left(G[i] \cap G[j] \neq \emptyset | G[j] = g\right)^2 \;=\; \frac{1}{2^a - 1} \sum_{g \subset \Sigma, g \neq \emptyset} \mathbb{P}\left(G[i] \cap g \neq \emptyset\right)^2, \end{aligned}$$

where for the last identity we have used that $G[j]$ is uniformly distributed, and that the generalized characters are i.i.d. Next, switching to the complementary event, observe that the probability of the event $G[i] \cap g = \emptyset$ depends only on the size of g. Moreover, if $|g| = \gamma$ then $G[i]$ can be any of $(2^{a-\gamma} - 1)$ subsets of Σ to have $G[i] \cap g = \emptyset$. Putting this all together, we find that

$$\begin{aligned} c_3 &= \frac{1}{2^a - 1} \sum_{\gamma=1}^{a} \binom{a}{\gamma} \left(1 - \frac{2^{a-\gamma} - 1}{2^a - 1}\right)^2 \\ &= \frac{4^a}{(2^a - 1)^3} \sum_{\gamma=0}^{a} \binom{a}{\gamma} \left(1 - \frac{1}{2^\gamma}\right)^2 \\ &= \frac{4^a}{(2^a - 1)^3} \left(\sum_{\gamma=0}^{a} \binom{a}{\gamma} - 2 \sum_{\gamma=0}^{a} \binom{a}{\gamma} \frac{1}{2^\gamma} + \sum_{\gamma=0}^{a} \binom{a}{\gamma} \frac{1}{4^\gamma} \right) \\ &= \frac{4^a}{(2^a - 1)^3} \left(2^a - 2 \left(\frac{3}{2}\right)^a + \left(\frac{5}{4}\right)^a \right) \;=\; \frac{8^a - (2)6^a + 5^a}{(2^a - 1)^3}, \end{aligned}$$

as claimed.

Finally, note that $c_2 = c_3 = 1$ when $a = 1$. To complete the proof of the lemma, it remains to be shown that $c_2^2 < c_3$ for $a > 1$. Indeed, if $a > 1$ then

$$\frac{c_3}{c_2^2} - 1 \;=\; \frac{10^a - 9^a - 8^a + 2 \cdot 6^a - 5^a}{(4^a - 3^a)^2}.$$

One can check using a computer that the right-hand side above is strictly positive for $1 < a < 8$. Instead, for $a \geq 8$, we can use the Binomial theorem to assert that $10^a = (9+1)^a > 9^a + a9^{a-1}$, hence

$$\frac{c_3}{c_2^2} - 1 \;>\; \frac{a9^{a-1} - 8^a + 2 \cdot 6^a - 5^a}{(4^a - 3^a)^2} \geq \frac{a9^{a-1} - 8^a}{(4^a - 3^a)^2} \geq \frac{a8^{a-1} - 8^a}{(4^a - 3^a)^2} \geq 0.$$

This completes the proof of the theorem. □

Figure 3.4 displays the parameters c_2 and c_3 plotted with respect to the alphabet size.

3.5 Highly Likely Bounds on Memory Consumption

The first result that we present shows that the size of prefixes matching with substrings is limited with high probability. This result is similar to what was found in the probabilistic analysis

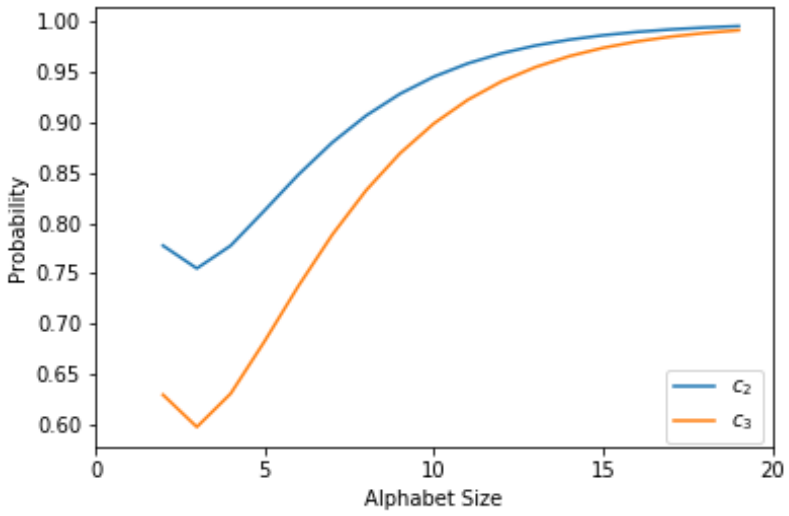

Figure 3.4: **Probabilities c_2 and c_3 in terms of the alphabet size when generalized characters are drawn uniformly at random among all non-empty subsets of Σ.**

by Aitmous et al. [2]; however, random generalized strings do not necessarily fit into the framework their analysis depends on.

The proceeding theorem leads to the conclusion that, if one fixes the generalized string under consideration, there is a cap on the number of unique failure functions (and thus states) a depth can have with high probability. As such, the DFA constructed for a fixed generalized string grows linearly with high probability. Although this gives insight for the rate at which a particular DFA grows when being constructed, this is not to say that DFA grow linearly as we increase the length of the generalized string. The size of a DFA as one varies the length of the generalized string will be addressed in the second half of this section, using the previous result. In particular, a bound for the size of the automaton will be given that is true with high probability.

Theorem 3.1. *Assume that* $0 < c_3 < c_2 < 1$. *For a given* $\epsilon > 0$, *define* $K := (1+\varepsilon) \cdot \log_{1/c_2}(n)$ *and* $k := \lceil K \rceil$. *Then, as* n *tends to infinity, the probability of the event* $G[:k] \cap G[i : i+k-1] = \emptyset$ *for all* $2 \leq i \leq (n-k+1)$ *converges to one.*

Proof. To start, define the random variable

$$Y_k \coloneqq \sum_{i=2}^{n-k+1} [\![G[:k] \cap G[i:i+k-1] \neq \emptyset]\!].$$

By Markov's inequality, or the first moment method:

$$\mathbb{P}(Y_k > 0) \leq \mathbb{E}[Y_k] = \sum_{i=2}^{n-k+1} \mathbb{P}\left(G[:k] \cap G[i:i+k-1] \neq \emptyset \right).$$

Let $t_i \coloneqq k - |\{1,\ldots,k\} \cap \{i,\ldots,i+k-1\}|$. By Lemma 3.0.3, to better characterize the probabilities on the right-hand side above we consider three cases: (1) $t_i = k$ (i.e. no overlap), (2) $2t_i \geq k$ (i.e. overlap is less than half of each substring), (3) $2t_i < k$ (i.e. overlap is greater than or equal to half of each substring). To address each of these, we divide the terms in the sum of $\mathbb{E}[Y_k]$ into $\mathcal{C}_1$, $\mathcal{C}_2$, and $\mathcal{C}_3$ which correspond to case 1, case 2, and case 3 respectively so that $\mathbb{E}[Y_k] = (\mathcal{C}_1 + \mathcal{C}_2 + \mathcal{C}_3)$. We now address one case at a time.

Case 1. Since the generalized characters in G are i.i.d. and $k \geq K$, we find that

$$\begin{aligned} \mathcal{C}_1 &= \sum_{i=k+1}^{n-k+1} \mathbb{P}\left(G[:k] \cap G[i:i+k-1] \neq \emptyset \right) \\ &= (n-2k+1)\, c_2^k \\ &\leq (n-2\,K+1)\, c_2^K \\ &= \frac{n - 2(1+\varepsilon)\log_{1/c_2}(n) + 1}{n^{1+\varepsilon}} \;=\; \mathcal{O}\left(n^{-\varepsilon}\right). \end{aligned}$$

Case 2. By part (1) in Lemma 3.0.3,

$$\begin{aligned} \mathcal{C}_2 &= \sum_{i=\lceil \frac{k+1}{2} \rceil + 1}^{k} \mathbb{P}\left(G[:k] \cap G[i:i+k-1] \neq \emptyset \right) \\ &\leq \sum_{t=\lfloor \frac{k}{2} \rfloor + 1}^{k} c_2^{2t-k} c_3^{k-t} \;=\; \left(\frac{c_3}{c_2}\right)^k \sum_{t=\lfloor \frac{k}{2} \rfloor + 1}^{k} \left(\frac{c_2^2}{c_3}\right)^t. \end{aligned}$$

We now consider two mutually exclusive scenarios. First, if $c_2^2 < c_3$ then $\mathcal{C}_2 = \mathcal{O}((c_3/c_2)^k)$ and

hence $\mathcal{C}_2 = o(1)$ because $c_3 < c_2$ by assumption. Instead, if $c_2^2 \geq c_3$ then

$$\begin{aligned}
\mathcal{C}_2 &\leq k\left(\frac{c_3}{c_2}\right)^k \left(\frac{c_2^2}{c_3}\right)^k \\
&= kc_2^k \\
&\leq \lceil (1+\varepsilon)\log_{1/c_2}(n)\rceil c_2^{(1+\varepsilon)\log_{1/c_2}(n)} \\
&\leq \frac{(1+\varepsilon)\log_{1/c_2}(n)+1}{n^{(1+\varepsilon)}} \;=\; \mathcal{O}\left(\frac{\log(n)}{n^{1+\varepsilon}}\right).
\end{aligned}$$

Putting this all together, we find that $\mathcal{C}_2 = o(1) + \mathcal{O}\left(\frac{log(n)}{n^{1+\varepsilon}}\right)$.

Case 3. Recall that

$$\mathcal{C}_3 = \sum_{i=2}^{\lceil \frac{k+1}{2}\rceil} \mathbb{P}\left(G[:k] \cap G[i:i+k-1] \neq \emptyset\right).$$

Due to part (2) in Lemma 3.0.3, we find that

$$\mathcal{C}_3 \leq \sum_{t=1}^{\lfloor \frac{k}{2}\rfloor} c_2^{r\lfloor \frac{d+2}{2}\rfloor + (t-r)\lfloor \frac{d+1}{2}\rfloor} \leq \sum_{t=1}^{\lfloor \frac{k}{2}\rfloor} c_2^{t\lfloor \frac{d+1}{2}\rfloor}.$$

But note that for that each term of the summation we have that

$$c_2^{t\lfloor \frac{d+1}{2}\rfloor} \leq c_2^{t\left(\frac{d+1}{2}-\frac{1}{2}\right)} = c_2^{\frac{t}{2}d} = c_2^{\frac{t}{2}\lfloor \frac{k}{t}\rfloor} \leq c_2^{\frac{t}{2}\left(\frac{k}{t}-\frac{t-1}{t}\right)} = c_2^{\frac{k}{2}-\frac{t}{2}+\frac{1}{2}}.$$

Finally, relating this back to the total sum, we find that:

$$\begin{aligned}
\mathcal{C}_3 &\leq \sum_{t=1}^{\lfloor \frac{k}{2}\rfloor} c_2^{\frac{k}{2}-\frac{t}{2}+\frac{1}{2}} \\
&\leq \left(\frac{k}{2}\right) c_2^{\frac{k}{2}-\frac{k}{4}+\frac{1}{2}} \\
&\leq \left(\frac{\lceil K\rceil}{2}\right) c_2^{\frac{K}{4}+\frac{1}{2}} \\
&= \left\lceil \left(\frac{1+\varepsilon}{2}\right)\log_{1/c_2}(n)\right\rceil c_2^{\left(\frac{1+\varepsilon}{4}\right)\log_{1/c_2}(n)+\frac{1}{2}} \;=\; \mathcal{O}\left(\frac{\log(n)}{n^{\frac{1+\varepsilon}{4}}}\right).
\end{aligned}$$

At last, we can combine all three cases together to obtain that

$$\mathbb{E}[Y_k] = \mathcal{O}\left(n^{-\varepsilon}\right) + o(1) + \mathcal{O}\left(\frac{\log(n)}{n^{1+\varepsilon}}\right) + \mathcal{O}\left(\frac{\log(n)}{n^{\frac{1+\varepsilon}{4}}}\right) = o(1).$$

Therefore, $\lim_{n\to\infty} \mathbb{P}(Y_k > 0) = 0$ as claimed. □

We can now link this result regarding substrings within G to states within a depth of the minimal DFA.

Corollary 3.1.1. *Define* $k := \lceil (1+\varepsilon) \cdot \log_{1/c_2}(n) \rceil$, *with* $\epsilon > 0$ *fixed. Let* S_i *be the number of states in depth* i *of the minimal DFA recognizing* $\Sigma^* G$, *with* G *of length* n *formed by i.i.d generalized characters such that* $0 < c_3 < c_2 < 1$. *Then, as* n *tends to infinity, the probability of the event* $S_i \leq \sum_{j=0}^{k-1} S_j$ *for all* $0 \leq i \leq n$ *converges to one.*

Proof. From Algorithm 2 we know that S_j is equivalent to the unique failure transitions that appear in depth j. Furthermore, from Theorem 3.1 we know that any substring of length k has low probability of matching with a prefix. Since this probability is monotonically decreasing with respect to the substring length, anything larger than this is also unlikely to occur.

Recall that the failure function is related to the longest suffix that is also a prefix. For this reason, with high probability there is no failure function that maps to a state in depth k or past it (Lemma 3.0.2). Thus, the number of possible unique failure functions that we can have is limited by $S = \sum_{j=0}^{k-1} S_j$.

□

To demonstrate the impact of this result, we look at synthetic data and how the consumption of memory grows during construction. In particular, 10,000 generalized strings of a fixed length, n, were drawn from the uniform distribution. The corresponding minimal DFA were constructed, the number of states at each depth noted, and the average trend plotted alongside $k = \log_{1/c_2}(n)$ (see Proposition 3.0.2). While in any one realization there will almost surely be a significant amount of fluctuation in the number of states per depth, the number of states after k will be capped at some constant with high probability. This explains why the average number of states per depth levels out (Figure 3.5), and why the overall trend of any given realization looks approximately linear near the end of construction (Figure 3.6).

It is important to note that so far our findings apply to generalized strings of a fixed size. It may not necessarily be the case, however, that the total number of states in these minimal

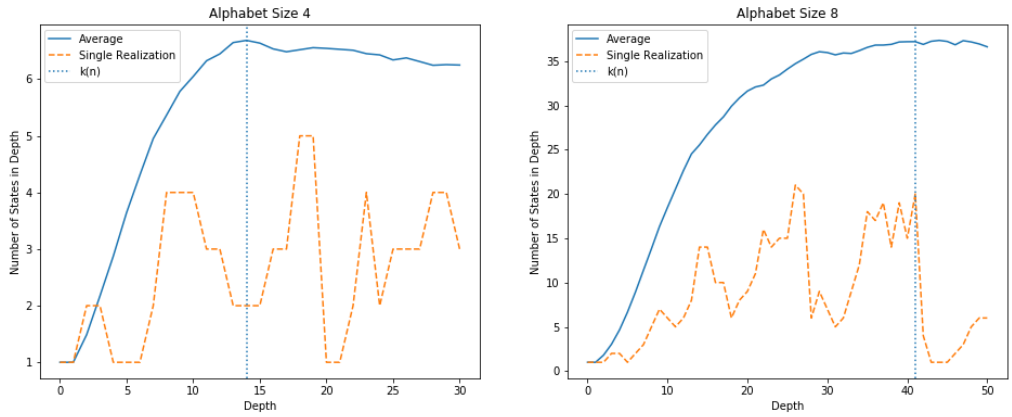

Figure 3.5: **Average number of states in each depth of the minimal automaton.** The solid line is the average number, the dashed, orange line shows the number of states versus depth for one particular DFA, and the dotted, blue line shows $k(n) = \log_{1/c_2}(n)$. The left plot assumed an alphabet size of 4 and generalized string length of $n = 35$, whereas the right plot has an alphabet size of 8 and a generalized string length of $n = 50$. In both cases, the generalized strings were drawn from the uniform distribution.

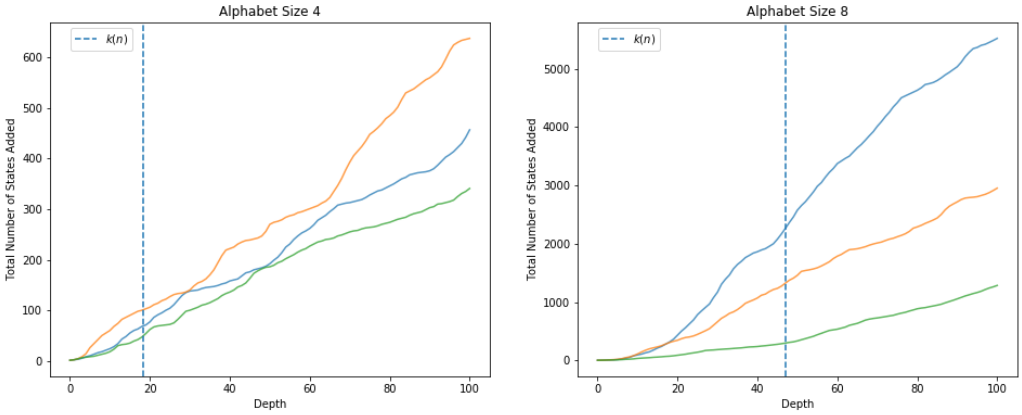

Figure 3.6: **Cumulative number of states in terms of depth during construction of the minimal automaton.** The left plot assumed an alphabet size of 4 and the right of 8. For each, three random generalized strings of length $n = 100$ were randomly picked according to the uniform distribution. The dotted blue line shows $k(n) = \log_{1/c_2}(n)$.

automata grows linearly with the length n of the associated generalized string (although, in the next chapter, we claim that the expected total size may grow linearly in some cases). This is because the rate of growth during construction may increase as n increases. That being said, we can leverage the previous results to obtain a polynomial bound for the total number of states with

respect to generalized string length that is true with overwhelmingly high probability when n is large.

Theorem 3.2. *Fix $\epsilon > 0$. For a generalized string G of length n with i.i.d generalized characters such that $0 < c_3 < c_2 < 1$, the total number of states in its associated minimal DFA is bounded by $n^{1+(1+\epsilon)\cdot log_{1/c_2}(2)}$, with a probability converging to one as n tends to infinity.*

Proof. Let S_i be the number of states in depth i of the minimal DFA associate with a generalized string of length n. Define $T_n := \sum_{j=0}^{k(n)-1} S_j$, with $k(n) = \lceil (1+\epsilon)\log_{1/c_2}(n) \rceil$. By Corollary 3.1.1, the total number of states is bounded by $n \cdot T_n$. By a logic similar to that used in Proposition 3.0.1, the number of state can at most double with each depth in the automaton. However, this trend cannot progress past depth $k(n)$ with high probability. Therefore $T_n \leq 2^{k(n)-1}$, with a probability converging to one as n tends to infinity. As a result, the total number of states is bounded by $n \cdot 2^{k(n)-1} \leq n^{1+(1+\epsilon)\cdot log_{1/c_2}(2)}$ with a probability converging to one as n tends to infinity. □

Although probabilistic in nature, this new polynomial bound represents a significant improvement over the deterministic (but exponential) bound given in Proposition 3.0.1. This can be appreciated in Figure 3.7 for an alphabet of size 3 and generalized strings of lengths $2 \leq n \leq 15$.

Unfortunately, the polynomial bound in Theorem 3.2 is never sub-quadratic in the uniform case because $c_2 > 0.5$ (see Figure 3.4). Moreover, it may not be practical for large alphabets because c_2 quickly approaches 1 as a increases, driving $\log_{1/c_2}(2)$ to infinity (see Figure 3.8). Likewise, the polynomial upper bound may be impractical for other schemes in which c_2 is close to 1.

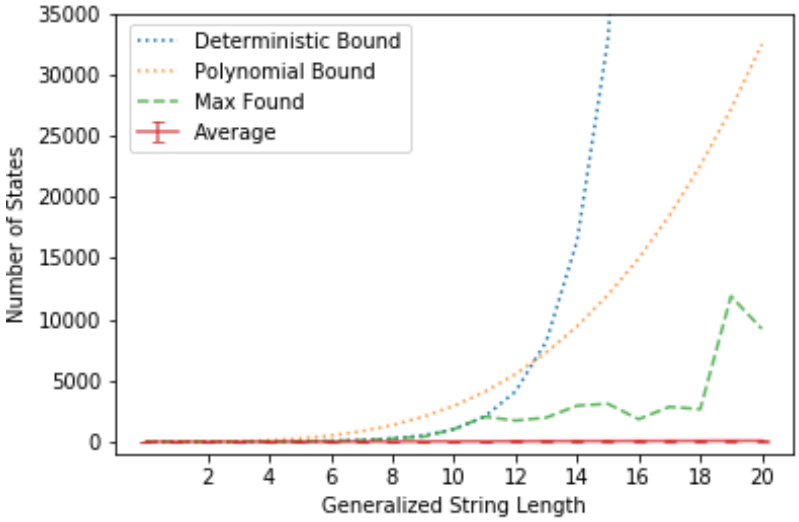

Figure 3.7: **Performance of the polynomial but probabilistic bound in Theorem 3.2 for an alphabet of size 3.** At each length, 10,000 minimal DFA associated with random generalized strings of that length were constructed. The maximum number of total states seen in any DFA, as well as the average number of total states across the 10,000 DFA are plotted. As an approximation of the polynomial upper bound, we set $\epsilon = 0$ and found the degree on n to be $3.467\ldots$.

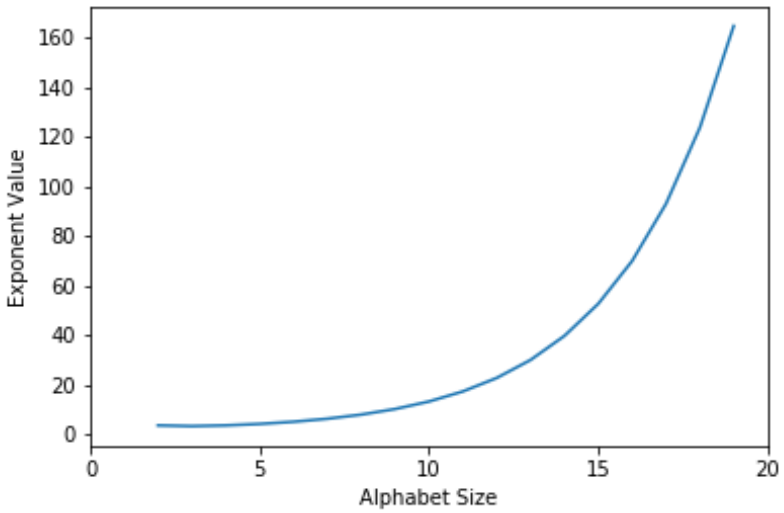

Figure 3.8: **Depiction of how the exponent of the probabilistic upper bound grows with the size of the alphabet.** To approximate this trend we set $\epsilon = 0$, i.e. the above shows $1 + \log_{1/c_2}(2)$.

Chapter 4

Claims and Heuristics for Future Work

In this chapter, we provide evidence as to why the expected number of states in a minimal DFA can grow linearly with respect to the length of the generalized string. For this, we introduce the concept of *signature* for a state and use this concept to form a particle-based model. Under a couple assumptions, this model can be leveraged to show the aforementioned linear growth in expected number of states. Furthermore, this argument reveals a method to estimate the slope of the growth.

4.1 Signatures

Consider a generalized string G, the corresponding minimal automaton formed via Algorithm 2: $M^*_{ACA} = (Q, \Sigma, q_0, \delta, F)$. The signature, $sig(q)$, of state $q \in Q$ is defined to be the tuple containing—in reverse order—the depths visited by the unique path from q to q_0 composed entirely of failure edges. Since this path will always start at q and end at q_0, the signature must have the form $(0, \ldots, depth(q))$. In particular, $sig(q_0) = (0)$. To fix ideas, see Figure 4.1.

The signature gives crucial insight about suffixes of the generalized string matching with prefixes. For example, suppose that a state q at depth 6 has the signature $(0, 1, 3, 4, 6)$, and let s be a string of length 6 such that $\delta(q_0, s) = q$. From the fact that there is a failure edge going from q to depth 4, it is apparent that $s[3:] \in G[:4]$. Following the same logic, we find that $s[4:] \in G[:3]$ and $s[6] \in G[1]$. This property is described with more generality in the proceeding lemma.

Lemma 4.0.1. *Consider a state $q \in Q$ and a string $s \in \Sigma^*$ such that $\delta(q_0, s) = q$. For $0 < x < depth(q)$, $x \in sig(q)$ if and only if $s[|s| - x + 1 :] \in G[: x]$.*

Proof. ($\Rightarrow$) Fix a state q and an $x \in sig(q)$ such that $0 < x < depth(q)$. Where $n \geq 1$ is the number of entries that are greater than x, we show the statement using an inductive argument in n.

For the base case with $n = 1$, the only entry greater than x is $depth(q)$. In other words, the failure of state q is in depth x, implying that there is an x-length suffix of s that matches with a prefix in G, i.e. $s[|s| - x + 1 :] \in G[: x]$. Next, assume the statement holds for certain $n \geq 1$. Let $y > x$ be the entry in the signature of q with n other entries greater than it. By the inductive hypothesis, we must have $depth(\delta(q_0, s[|s| - y + 1 :])) = y$. Since the signature of q implies that there is a failure edge in the path of failures that goes from depth y to depth x, there must be an x-length suffix of $s[|s| - y + 1 :]$ that matches with a prefix in G, i.e. $s[|s| - x + 1 :] \in G[: x]$ as claimed.

($\Leftarrow$) Now suppose that for a valid x, $s[|s| - x + 1 :] \in G[: x]$, i.e. there is an x-length suffix of s that matches with some prefix of G. For the sake of contradiction, suppose that x is not in $sig(q)$. Let a and b be the largest and smallest entries of $sig(q)$, respectively, such that $a < x < b$. By the first part of the proof, and the fact that the path corresponding to the signature has a failure from depth b to depth a, we know that $s[|s| - a + 1 :]$ is the largest suffix of $s[|s| - b + 1 :]$ that matches with a prefix of G. However, this is not possible because $s[|s| - x + 1 :]$ is a suffix of $s[|s| - b + 1 :]$ that matches with a prefix in G, and $s[|s| - x + 1 :]$ is longer than $s[|s| - a + 1 :]$. This contradiction implies that $x \in sig(q)$. □

As shown next, signatures uniquely identify their state, hence the name "signature."

Lemma 4.0.2. *Any state in the minimal automaton M^*_{ACA} recognizing $\Sigma^* G$ is uniquely identified by its signature.*

Proof. We have already mentioned that for any given state there is only one path composed of failure edges that will lead to the start state. This is because each state (besides the start state) has one, and only one, value for the failure function. Thus each state can only have one signature.

It remains to be shown that no two states have the same signature. We do this by induction on signature sizes.

Clearly, since (0) is the only possible signature of size 1, and q_0 is the only state in M^*_{ACA} at depth 0, it follows that $|sig(q)| = 1$ if and only if $q = q_0$. Now assume that all signatures of size $k \geq 1$ uniquely identify their corresponding state in the minimal automaton. Suppose there are states $p, q \in Q$ such that $sig(p) = sig(q)$ and $|sig(p)| = (k+1)$. From the definition of signature, it is immediate that $depth(q) = depth(p)$. Furthermore, the failure edges of p and q must point to states that have the same k-sized signature. Since, by the inductive hypothesis, there can only be one state with this signature, it follows that $f(p) = f(q)$. In particular, because Algorithm 2 merges states at the same depth with identical failures, we must have $p = q$. □

Finally, we show that signatures build off of parent's signatures. The intuition behind our next result can be elucidated via an example. For instance, if after reading in $ACBBCBD$ one knows that the suffix $BCBD$ matches with a prefix in G, it must be true that BCB also matches with a prefix in G. More generally, if a suffix of length x matches with a prefix after reading in n characters, it must be true that there was a matching suffix of length $(x-1)$ after reading in $(n-1)$ characters. Thus, suffixes must be iteratively built up, one letter at a time.

Lemma 4.0.3. *For the minimal automaton M^*_{ACA} recognizing $\Sigma^* G$, consider a state $q \in Q$. If p is a parent of q and $x \in sig(q)$ is non-zero, then $(x-1) \in sig(p)$.*

Proof. Note that this statement is trivial when $x = 1$ or $x = depth(q)$. Therefore, we can assume without loss of generality that $1 < x < depth(q)$. Let s be a string of length $depth(q)$ such that $\delta(q_0, s) = q$ and $\delta(q_0, s[: depth(q) - 1]) = p$. Since $x \in sig(q)$, by Lemma 4.0.1 the x-length suffix of s matches with some prefix of G, i.e. $s[depth(q) - x + 1 :] \in G[: x]$. But this immediately implies that $s[depth(q) - x + 1 : depth(q) - 1] \in G[: x - 1]$, i.e. $s[: depth(q) - 1]$ has an $(x-1)$-length suffix that is a prefix of G. Since $\delta(q_0, s[: depth(q) - 1]) = p$, again using Lemma 4.0.1, this implies that $(x-1) \in sig(p)$. □

To see Lemma 4.0.3 in practice, one can refer to Figure 4.1. One such example can be seen by signature $(0, 2, 3)$ and the parent signature $(0, 1, 2)$, for which $(2 - 1) = 1$ and $(3 - 1) = 2$ is in the parent signature. This property gives rise to the model described in the next section.

4.2 A particle-based model

Lemma 4.0.3 shows that signatures in depth n expand upon signatures in depth $(n-1)$. Using this for inspiration, we refer to signatures of length at least three as *particles*, and say that two particles are in a *parent-child relationship* if and only if their corresponding states in the automaton (see Lemma 4.0.2) are also in a parent-child relationship.

Let Z_n be the number of particles in the automaton at depth n. We focus on signatures of length three because the case $sig(q) = (0, depth(q))$, for some state q, represents the scenario in which there are no suffixes that match with a prefix. In particular, this case can only happen once per depth, so Z_n is at most one less than the total number of signatures in depth n.

We note that the particle model does not naturally correspond to a branching process. This is because a particle may have multiple parents, or no parent at all. For example, a signature of the form $(0, 1, n)$ might not stem from a particle because its parent may be the signature $(0, n-1)$. To precisely capture this anomaly, define X_n to be the indicator of the event that signature $(0, 1, n)$ appears in depth n. Moreover, define Y_n to be the average number of offspring (excluding the offspring $(0, 1, n + 1)$) each particle has at depth n. Put more simply,

$$Y_n \coloneqq \begin{cases} \frac{Z_{n+1} - X_{n+1}}{Z_n} & , \text{ when } Z_n > 0; \\ 0 & , \text{ when } Z_n = 0. \end{cases}$$

In particular:

$$Z_n = X_n + Y_{n-1} \cdot Z_{n-1} \tag{4.1}$$

An example with this recursion can be seen in Figure 4.1.

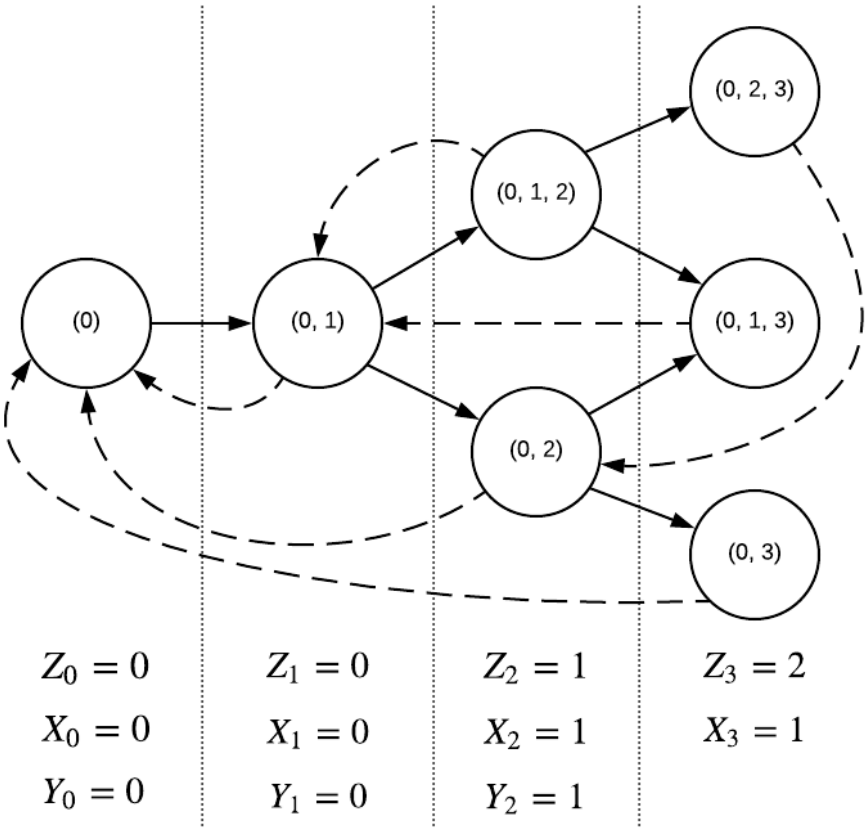

Figure 4.1: **A minimal DFA where the random variables in equation (4.1) are stated explicitly.** Each state is labelled with its corresponding signature. The states causing X_2 and X_3 to take the value 1 are the ones labelled with the signatures $(0, 1, 2)$ and $(0, 1, 3)$. Note that Y_3 is not defined because, in this example, depth 3 is the last depth.

4.3 Conjectured Properties and Expected Linear Growth

For the remainder of the chapter we will assume that the following properties hold:

(A1) $\mathbb{E}[X_n]$ and $\mathbb{E}[Y_n]$ converge to constants μ_x and μ_y, respectively, as n tends to infinity.

(A2) There exists a finite constant $C > 0$ such that $|\mathrm{Cov}(Y_n, Z_n)| \leq C$, for all n.

4.3.1 Evidence of Assumptions

Although assumptions (A1)-(A2) cannot be proven mathematically at this time, we present heuristics and synthetic data to support their validity.

First we address assumption (A1). One can develop an intuition for this assumption by

considering longest suffixes that match with a prefix in the (random) generalized string. Specifically, for a given ℓ, imagine wanting to determine the expected length of the longest suffix of $G[:\ell]$ that matches with a prefix of this generalized string. One would expect that, at a certain point, increasing ℓ does not have much of an effect on that expected value. For example, changing ℓ from 100 to 200 is unlikely to change the expected value by much since it is highly unlikely that one sees a suffix of $G[:200]$ of length over 100 that matches with a prefix of $G[:200]$.

Relating this back to X_n and Y_n, one can see that these quantities are influenced by the behavior of the failure function. This is straightforward to see for X_n since X_n takes the value 1 if there is a failure edge leading back to depth 1 and 0 otherwise. Y_n is also related to the failure function since the number of unique failure edges determines the number of particles and offspring at depth n. In particular, the value of Y_n is the number of failure edges stemming from depth $(n+1)$ and leading to a depth of at least 2 divided by the number of failure edges stemming from depth n and leading to a depth of at least 1. It is therefore not unreasonable to assume that, similar to the expected length of a suffix matching with a prefix, $\mathbb{E}[X_n]$ and $\mathbb{E}[Y_n]$ also converge to a constant.

Figure 4.2 provides evidence from synthetic data that supports assumption (A1). To ensure that the observed μ_x and μ_y are not dependent on the generalized string length picked, data was generated for generalized string lengths of 10, 25, and 50. As one can see, the trends are nearly identical regardless of length.

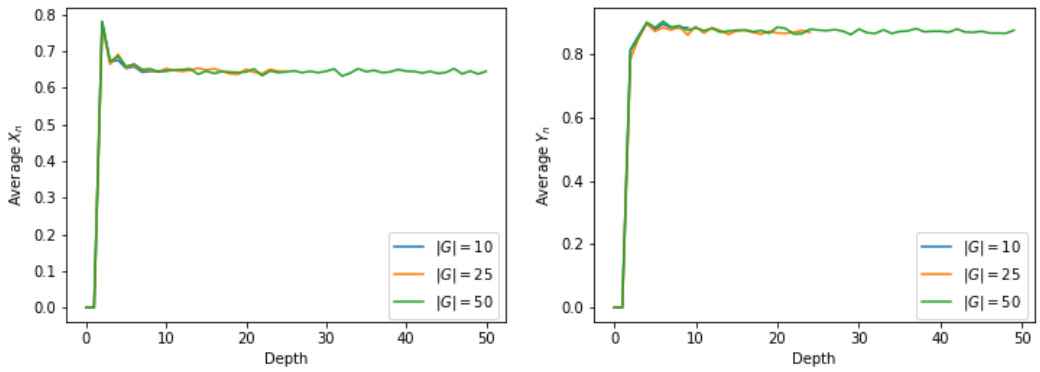

Figure 4.2: **Sample averages for X_n and Y_n in terms of n .** For each of the lengths considered, 10,000 generalized strings were drawn according to the uniform distribution with an alphabet of size 4.

Given assumption (A1), it seems reasonable to assume that (A2) is also true. Because the synthetic data exhibits stabilization, it seems unlikely that we would see a case in which the covariance blows up towards infinity. Indeed, Figure 4.3 shows how the covariance between Y_n and Z_n tends to stabilize as one looks deeper within the automaton.

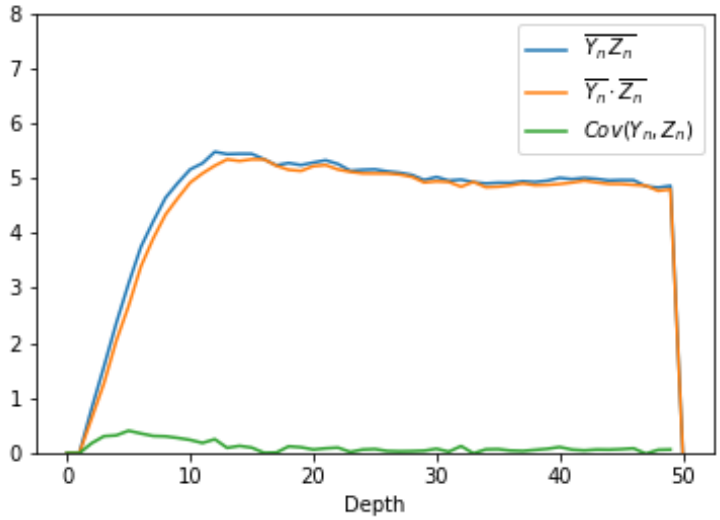

Figure 4.3: **Sample covariance between Y_n and Z_n in terms of n.** Along with this, the sample averages $\overline{Y_n Z_n}$ and $\overline{Y_n} \cdot \overline{Z_n}$ are shown for reference. For each of the lengths considered, 10,000 generalized strings were drawn according to the uniform distribution with an alphabet of size 4.

4.3.2 Linear Expected Growth of Automaton

Assuming that (A1) and (A2) hold, the fact that the automaton grows linearly in expectation should not come as a surprise. In Chapter 3 we saw that with overwhelmingly high probability the minimal DFA grows linearly during construction for a fixed but large length of the generalized string. That being said, there was a concern that increasing the length of the generalized string could also increase its rate of growth. Here, however, assumption (A1) implies that the expected size of the automaton grows at fixed rate, independent of the generalized string length.

Theorem 4.1. *Let G be a random generalized string of length ℓ with i.i.d generalized characters. If (A1) and (A2) are true, with $\mu_y < 1$, then the expected number of states in M^*_{ACA} grows linearly*

with ℓ.

Proof. Due to equation (4.1), we have that $\mathbb{E}[Z_n] = \mathbb{E}[X_n] + \mathbb{E}[Y_{n-1}Z_{n-1}]$. But, by assumption (A2), there exists a constant $C > 0$ such that $|\mathrm{Cov}(Y_n, Z_n)| \leq C$ for all n. As a result:

$$\mathbb{E}[Z_n] \leq \mathbb{E}[X_n] + \mathbb{E}[Y_{n-1}] \cdot \mathbb{E}[Z_{n-1}] + C.$$

By assumption (A2), there is a finite constant $M_x > 0$ such that $\mathbb{E}[X_n] \leq M_x$ for all $n \geq 0$ because $\mathbb{E}[X_n]$ is a convergent sequence. Likewise, fixing a constant M_y such that $\mu_y < M_y < 1$, there exists an integer $N \geq 0$ such that $\mathbb{E}[Y_n] \leq M_y$ for all $n \geq N$. Thus $\mathbb{E}[Z_n] \leq M_x + M_y \cdot \mathbb{E}[Z_{n-1}] + C$, for $(n-1) \geq N$. One can now iteratively expand the right hand side in this inequality to obtain that

$$\mathbb{E}[Z_n] \leq M_y^{n-N} \cdot \mathbb{E}[Z_N] + (M_x + C) \sum_{j=0}^{n-N-1} M_y^j, \tag{4.2}$$

for all $(n-1) \geq N$. But note that $\mathbb{E}[Z_N]$ is finite because, according to Proposition 3.0.1, $Z_N \leq 2^N$. As a result, the sequence on the right-hand side above converges as n tends to infinity, and hence there is a finite constant $M > 0$ such that $E[Z_n] \leq M$ for all $n \geq 0$. The total number of expected states in the minimal automaton recognizing $\Sigma^* G$ is therefore bounded by $(M+1)\ell$. □

4.3.3 An Estimate of the Linear Growth's Slope

The inequality in equation (4.2) implies that

$$\limsup_{n \to \infty} \mathbb{E}[Z_n] \leq \frac{M_x + C}{1 - M_y}.$$

When providing evidence to justify assumption (A1), we saw in Figure 4.2 that $\mathbb{E}[X_n]$ and $\mathbb{E}[Y_n]$ converge very quickly. Furthermore, Figure 4.3 shows that the magnitude of $\mathrm{Cov}(Y_n, Z_n)$ remains relatively small. Thus, one might expect that a good estimate to this asymptotic upper bound is $\frac{\mu_x}{1-\mu_y}$. To test this, the estimates $\hat{\mu_x}$ and $\hat{\mu_y}$ can be found by averaging X_n and Y_n at a sufficiently large depth. These estimates were used to calculate the approximate upper bound shown in Figure 4.4.

As one can see, this value serves as a decent estimate to the greatest number of particles in any depth. It therefore stands to reason that $1 + \frac{\mu_x}{1-\mu_y}$ should be a good way to estimate the slope

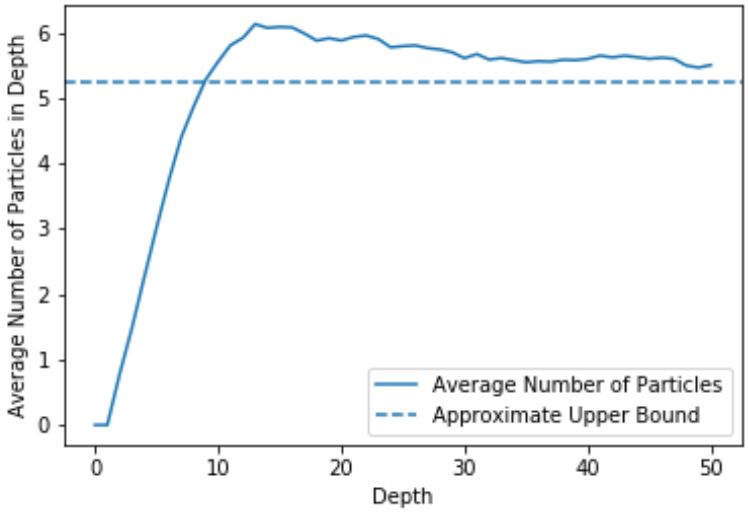

Figure 4.4: **A comparison between the average number of particles at each depth and the approximate upper bound.** 10,000 generalized strings were drawn from the uniform distribution with alphabet size of 4.

of the expected growth in the total number of states (plus one since number of particles differs from number of states by at most one). Indeed, Figure 4.5 shows that this estimated slope is quite close to the trend of total average number of states. Furthermore, Figure 4.6 shows that the estimate remains reasonable for differing alphabet sizes.

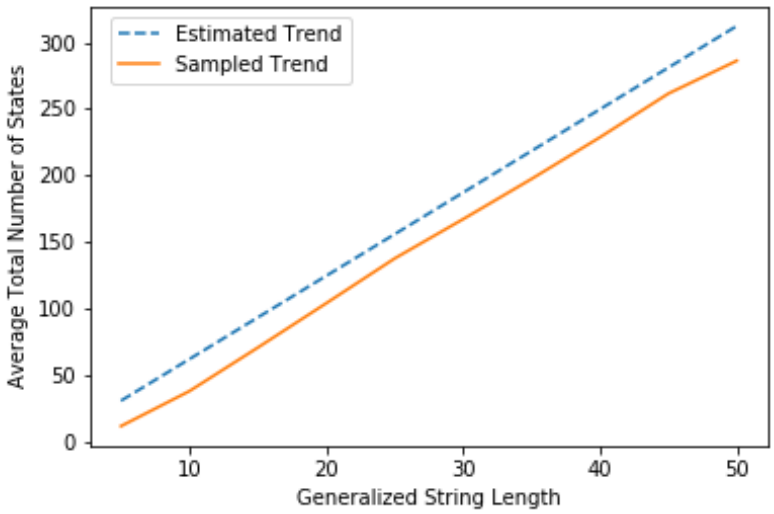

Figure 4.5: **A comparison between average total number of states and the estimated slope.** For lengths in increment of 5 from 5 to 50, 10,000 generalized strings were drawn according to the uniform distribution with alphabet size 4, and the average total number of states computed.

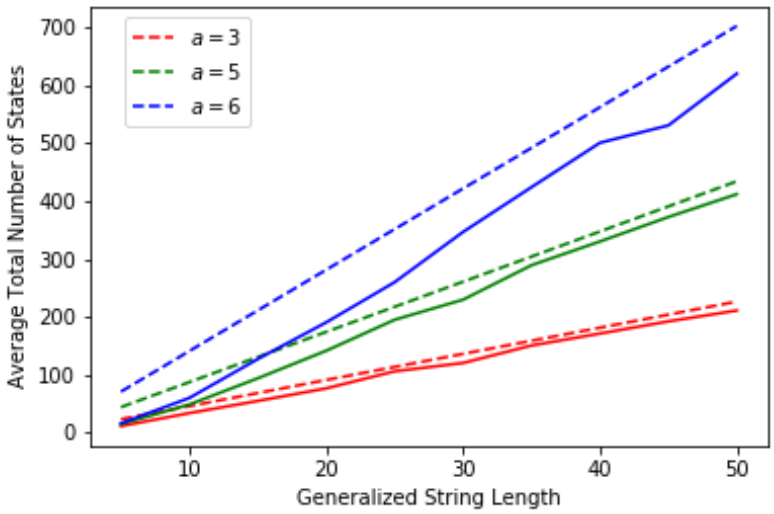

Figure 4.6: **Comparison between average total number of states and the estimated slope for three different alphabet sizes.** Each solid line represents the sample average for total states. For lengths in increment of 5 from 5 to 50, 1,000 generalized strings were drawn from the uniform distribution. The corresponding dotted lines of the same color show the estimated slope.

Chapter 5

Conclusion and Future Work

The main problem that we aimed to address in this thesis is how to deal with and account for large DFA corresponding to generalized strings. To do this, we first introduced a novel algorithm to construct the minimal DFA recognizing languages of the form $\Sigma^* G$ (Algorithm 2) for a generalized string G. This algorithm relies heavily on the Aho-Corasick automaton. In particular, the failure edges in the automaton are used to signify when two states should be merged. Although there exists other algorithms that construct the minimal automaton for generalized strings [6], a significant advantage of our algorithm is that the minimal automaton can be formed directly. Furthermore, our algorithm allows for the transition function to be encoded via goto and failure functions, further cutting down memory consumption.

Even with an algorithm that produces the minimal DFA, there is no guarantee that the produced automaton will be small enough to fit into computer memory. Our attention thus turned to finding an upper bound to the size of minimal DFA. Since the new algorithm is iterative in nature, it was used to aid the search of such an upper bound. We first showed that, when the generalized string has length n, the total number of states in the associated minimal DFA is bounded by 2^n—regardless of the reference alphabet (Proposition 3.0.1). To get a tighter upper bound, we introduced some randomness into the system. Specifically, we assumed that the generalized characters in G were i.i.d. Under this assumption, we found that there exists some $k = k(n)$ such that that the generalized string contains no substring of length k matching with a prefix with high probability (Theorem 3.1). Relating this back to the associated failure function of the minimal DFA, we found

a new bound that holds with high probability and grows polynomially rather than exponentially (Theorem 3.2). The degree of this polynomial depends on the alphabet size and the distribution of the generalized characters.

Finally, we used a particle-based model to argue that there may be cases in which the expected number of states in the minimal DFA grows linearly with the length of the generalized string (Theorem 4.1). Furthermore, we provided a way to estimate the rate of growth. These last two results depend, however, on certain assumptions. While evidence of the assumptions' validity was given via synthetic data, they are left unproven for now.

Bibliography

[1] Alfred V. Aho and Margaret J. Corasick. Efficient string matching: an aid to bibliographic search. Communications of the ACM, 18(6):333–340, 1975.

[2] Omar AitMous, Frédérique Bassino, and Cyril Nicaud. An efficient linear pseudo-minimization algorithm for Aho-Corasick automata. In Annual Symposium on Combinatorial Pattern Matching, pages 110–123. Springer, 2012.

[3] Jan Daciuk, Stoyan Mihov, Bruce W. Watson, and Richard E. Watson. Incremental construction of minimal acyclic finite-state automata. Computational Linguistics, 26(1):3–16, 2000.

[4] John Hopcroft. An n log n algorithm for minimizing states in a finite automaton. In Theory of Machines and Computations, pages 189–196. Elsevier, 1971.

[5] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. Introduction to Automata Theory, Languages, and Computation. Addison–Wesley, 2nd edition, 2001.

[6] Tobias Marschall. Construction of minimal deterministic finite automata from biological motifs. Theoretical Computer Science, 412(8):922–930, 2011.

[7] Edward F. Moore. Gedanken-experiments on sequential machines. Automata Studies, 34:129–153, 1956.

[8] Dominique Revuz. Minimisation of acyclic deterministic automata in linear time. Theoretical Computer Science, 92(1):181–189, 1992.