Tolerant Testing of Regular Languages in Sublinear Time

by

Linda Gong

B.S. Computer Science and Engineering, Massachusetts Institute of Technology, 2020

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2021

Author . . . .
Department of Electrical Engineering and Computer Science
May 20, 2021

Certified by . . . .
Ronitt Rubinfeld
Edwin Sibley Webster Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by . . . .
Katrina LaCurts
Chair, Master of Engineering Thesis Committee

Tolerant Testing of Regular Languages in Sublinear Time

by

Linda Gong

Submitted to the Department of Electrical Engineering and Computer Science
on May 20, 2021, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

A classic problem in *property testing* is to test whether a binary input word w is in regular language L. Such testers distinguish the case that w is in L from the case where w is ϵ-far from L (ϵ-far means that at least ϵ fraction of the bits in w must be modified to change w into a word in L. Otherwise, w is ϵ-close). When it is known that w is noisy, it can be useful to provide tolerant testers: algorithms that accept when w is δ-close and reject when w is ϵ-far, for $\delta < \epsilon$. We build on the work of Alon, Krivelevich, Newman and Szegedy [1] to provide a tolerant, constant time property tester for regular languages. Our main result is that given a regular language $L \in \{0,1\}^*$ and an integer n, there exists a randomized algorithm which *accepts* a word w of length n if it is δ-close ($\delta < \epsilon$) to a word in L and *rejects* with high probability if w is ϵ-far from a word in L. The algorithm queries polynomial in $\frac{1}{\epsilon}$ bits in w.

Thesis Supervisor: Ronitt Rubinfeld
Title: Edwin Sibley Webster Professor of Electrical Engineering and Computer Science

Acknowledgments

I would like to thank my advisor Prof. Ronitt Rubinfeld for all her guidance over my years at MIT. Ronitt introduced me to the first research paper that I ever read. Ronitt made algorithms seem less daunting; without her, I may have never viewed myself as capable of theoretical research. She helped me explore the field of sublinear algorithms and was supportive in every step of the problem-solving process.

I would also like to extend a hearty thanks to my mentor, collaborator and coauthor on this project – Talya Eden. Talya provided direction in the research and steered me away from many mathematical pitfalls. This paper certainly would not be what it is without Talya.

Thank you to the members of the Sublinear Algorithms research group. Through reading groups and friendship, these people have provided a warm environment, conducive to learning in the field.

Finally, thank you to my friends and family for supporting me through my five years at MIT. In particular, I would like to thank my mother and brother. Thank you for always believing me and providing me love when I need it most.

THIS PAGE INTENTIONALLY LEFT BLANK

Contents

THIS PAGE INTENTIONALLY LEFT BLANK

List of Figures

THIS PAGE INTENTIONALLY LEFT BLANK

Chapter 1

Introduction

The field of property testing relaxes the notion of a standard decision problem [13] [6]. A typical decision problem asks whether or not an input satisfies property P. First formulated by Rubinfeld and Sudan, a property testing algorithm for property P is given oracle access to the tested function f. The algorithm must then distinguish between the case that f has property P or that f is far from having property P [13] [6]. An input is said to be ϵ-far from P if ϵ-fraction of the input needs to be modified to obtain P. This relaxation of decision problems allows for very efficient algorithms with sublinear time and oracle query-complexity. There are many natural applications of property testing, especially in the context of PAC learning, program-checking, approximation algorithms and more [4] [13] [6]. Testing algorithms have been developed for functions, graphs, strings, geometric objects and more [6] [5].

In the following thesis, we refer to the above notion of property testing as *standard* property testing. Standard property testing is allowed to reject when an input is very close to a given property. For example, in the context of regular languages, Alon et al. [1] developed an algorithm that accepts when an input word w of length $|w| = n$ is in regular language L and rejects when ϵn bits need to be modified to obtain a word in L. However, this algorithm is allowed to reject when only one bit of the word w needs to be modified to obtain a word in L.

Thus, we consider a generalized notion of property testing that requires the tester to be *tolerant* with respect to inputs that are close to having a property [12]. Specifically, a tolerant property tester is required to accept inputs that are δ-close to having a property and reject inputs that are ϵ-far from having the property, $0 \leq \delta \leq \epsilon \leq 1$. Ideally, the tolerant property tester works for all choices of δ and ϵ with runtime sublinear in input size. Note that when $\delta = 0$, we achieve the standard notion of a property tester.

Tolerant property testing is a natural extension to standard property testing. Especially when it is known that the input is noisy, we may care for an algorithm that accepts inputs close to a given pattern. In this thesis, we focus on tolerant testing regular languages. Regular languages are exactly equivalent to the class of languages recognized by finite automata and state machines. In addition, regular languages are equivalent to those recognized by regular expressions (also known as regexp), which are commonly used in pattern matching programs. A tolerant testing algorithm for regular languages will accept all words for which at most δn bits need to be flipped in order to obtain a word in regular language L and will reject all words for which at least ϵn bits need to be flipped to obtain a word in L. The algorithm presented in this thesis is adaptable for choices of δ and ϵ for $\delta < 2^{-12}\frac{c\epsilon}{\log(\frac{1}{\epsilon})}$, where the constant c is a property of the regular language L.

1.1 Prior Work

This work is a continuation off the work done in [1]. The main result from that paper is a randomized algorithm that accepts a word w if it is in regular language L and rejects with probability at least $\frac{2}{3}$ if w is ϵ-far from L. The query complexity of their algorithm is $\tilde{O}(\frac{1}{\epsilon})$, which is optimal up to a poly-logarithmic factor in $\frac{1}{\epsilon}$. The authors make many observations about the periodicity and connectivity of deterministic finite automata (DFAs) that we will use in our paper. In addition, the authors introduce the very useful concept of "Admissible Triplets", which we make use of in our algorithm, too.

Subsequent work has either considered testers for more general classes of languages, such as context free languages [7] or attempts to make the algorithm more efficient with respect to the regular language's underlying automata (vs. input size) [10]. To our knowledge, the question of tolerantly testing regular languages has not been considered.

1.2 Contributions

In this thesis, we address the tolerant testability of formal languages in $\{0,1\}^*$. [1] For general references on languages, see [15] [8]. A language is a property which can be viewed as a set of strings or a sequence of Boolean functions $f_n : \{0,1\}^n \to \{0,1\}$, where $f_n^{-1}(1) = L \cap \{0,1\}^n = L_n$. Our main result shows that regular languages are tolerantly testable with parameters δ, ϵ, $\delta < 2^{-12}\frac{c\epsilon}{\log(\frac{1}{\epsilon})}$ with query complexity polyonomial in $O(\frac{1}{\epsilon})$. The c in the bound on δ is a constant and dependent on properties of the regular language L.

In Chapter 2, we specify and frame the problem by going over useful definitions and theorems. In Chapter 3, we provide two main ideas: one of them explores the gap between δ and ϵ for a single basic case and the other one expands the first idea to cover all automata. In Chapter 4, we provide algorithm and prove its correctness. Finally, in Chapter 5 we conclude and provide further steps.

[1]The $*$ operator is the Kleene Star

THIS PAGE INTENTIONALLY LEFT BLANK

Chapter 2

Preliminaries

In this chapter, we review some preliminary definitions and previously obtained theorems that are useful for our algorithm.

2.1 Standard Definitions

We start with the standard definition of a regular language, based on a finite automata

Definition 2.1.1. A *deterministic finite automata (DFA)* over $\{0,1\}$ with states $Q = \{q_1, .., q_m\}$ is given by a function $\delta : Q \times \{0,1\} \to Q$ together with a set $F \subseteq Q$. The state q_1 is called the start state. The states belonging to F are the accepting states and δ is the transition function.

Given an input word, the computation starts at q_1 and follows the transition functions to different states for each symbol in the word. An automata accepts an input if, after reading the entire input, it is in an accept state. Else, it rejects. In this paper, as in [1], we extend the transition function δ to $\{0,1\}^*$ recursively as follows: let γ denote the empty string, then

$$\begin{aligned} \delta(q,\gamma) &= q \\ \delta(q,u0) &= \delta(\delta(u,q),0) \\ \delta(q,u1) &= \delta(\delta(u,q),1) \end{aligned}$$

If an automata M is at position q and it processes string u, it ends up in state $\delta(q, u)$. Then, we say M accepts word u if $\delta(q_1, u) \in F$. Else, $\delta(q_1, u) \in Q \setminus F$. The language of an automata $L(M)$ is the set of words that it accepts.

Definition 2.1.2. A language is *regular* iff there exists a DFA that accepts it.

In this paper, we write $dist(w, L)$ to be the distance of word w from regular language L. This is the minimum number of bits in w that need to be flipped in order to make a word in L. We also write $dist(w, w')$ to be the distance of word w from word w'. This is the number of bits in w that need to be flipped to reach w'.

2.2 Useful Theorems

Note that every deterministic finite automata M has an underlying directed graph $G(M) = (V(G(M)), E(G(M)))$ where $V(G(M)) = Q$ and $E(G(M)) = \{(q_i, q_j) \mid \delta(q_i, 0) = q_j\} \cup \{(q_i, q_j) \mid \delta(q_i, 1) = q_j\}$. The *period* $g(G)$ of a directed graph G is the greatest common divisor of cycle lengths in G. If G is acyclic we set $g(G) = \infty$. Our algorithm relies on heavily on the properties of strongly connected components and periodicity. We reproduce the following lemma from below [1] (it is "Lemma 2.3" in the paper):

Lemma 2.2.1. Let $G = (V, E)$ be a nonempty, strongly connected directed graph with finite period g(G). Then, there exist a partition $V(G) = V_0 \cup ...V_{g-1}$ and a constant $m = m(G)$ which does not exceed $3|V|^2$ such that

1. For every $0 \leq i, j \leq g - 1$ and for every $u \in V_i$, $v \in V_j$ the length of every directed path from u to v in G is $(j - i) \mod g$;

2. For every $0 \leq i, j \leq g - 1$ and for every $u \in V_i, v \in V_j$ and for every integer $r \geq m$ if $r = (j - i) \mod g$, then there exists a directed path from u to v of length r

We will not reproduce the entire proof here, but the main observation is that any closed walk in G is the disjoint union of cycles (informally, imagine "taking out" cycles from the closed walk) and thus, the length of any closed walk is divisible by the period g [1]. This forms a disjoint partition of nodes based on the period and condition 1 follows quite naturally. The second condition follows from a well-known number theory result that for any set of positive integers $\{a_i\}$ whose gcd is g, there is a smallest number t such that for all integers $s \geq t$, s can be expressed as an integer linear combination of $\{a_i\}$. Moreover, we know that [9], t is smaller than the square of $\max(\{a_i\})$. Then, setting $m = 3|V|^2$ (to give enough time to traverse each cycle as many times as desired) provides condition 2. [1]

Definition 2.2.1. We call the constant m as defined in 2.2.1 the *reachability constant* of a strongly connected graph G. We assume that m is divisible by g.

In other words, we know that if a graph is strongly connected, we can start at any node and reach any other node with a sufficiently long enough path. This is a key component to our algorithm. We introduce another definition:

Definition 2.2.2. Given a word $w \in \{0,1\}^n$, a continuous sub-word or *run* w' of w starting at position i is called *feasible* for language $L(M)$ if there exists state q such that q is reachable from q_1 in $G(M)$ in exactly $i-1$ steps and if there is a path of length $n-(|w'|+i-1)$ in G from the state $\delta(q,w')$ to at least one of the accepting states. Otherwise w' is *infeasible.* [1]

Note that the feasibility of a run depends on the position i that it starts at. A run w' starting at index i is only *infeasible* if there are no viable prefixes (of length i) or suffixes of length $(n-(|w'|+i-1))$ that can be attached to i at all such that the DFA will accept. This means that if a word contains an infeasible run, it is not in the regular language L.

Finally, we provide the following definition [1] as the most simple case to consider:

Definition 2.2.3. An automata M is *essentially strongly connected* if

1. M has a unique accepting state q_{acc}

[1]For a more precise proof, please read [1]

2. M's set of states Q can be partitioned into disjoint sets C and D such that

    - $q_1, q_{acc} \in C$
    - The subgraph of $G(M)$ induced on C is strongly connected
    - No edges in $G(M)$ go from D to C (edges may go from C to D). D may be empty.

Essentially, an automata is *essentially strongly connected* if it contains a strongly connected component C and a set of garbage states D, where if the garbage states are reached, the input can no longer be accepted.

Definition 2.2.2 and Lemma 2.2.1 motivate our algorithm. We hope to show that if an input is δ-close, there are a limited number of short, infeasible runs it may have and if an input is ϵ-far, then it must have many short, infeasible runs. Thus, if we sample some amount of runs from our input and test them efficiently, we should be able to test w's approximate membership in L. The efficient tester is motivated by the periodicity of the strongly connected components of the automata. If our run w' starts and ends in positions that are length m away from the beginning and end of word w, we do not need to test all possible prefixes and suffixes to the run in order to see if w' is feasible.

What remains in the paper is prove that δ-close and ϵ-far result in a testable gap in the number of infeasible runs, show how we handle automata with multiple strongly connected components, and provide the algorithm for tolerant testability.

Chapter 3

Main Theorems

In this chapter, we show that regular languages can be tolerantly tested in the basic case where the automata for the language is essentially strongly connected. Then, we provide the framework to extend the testing to the general set of regular languages.

3.1 Gap in the Basic Case

In this thesis, we only consider testing nontrivial languages. This means that $L \cap \{0,1\}^n \neq \emptyset$. If the intersection is empty, there is already an algorithm to calculate the intersection of the DFAs and to test for emptiness of a DFA. These algorithms are polynomial in terms of the automata but constant in terms of the input size n.

Now, we present a lemma that provides a condition to distinguish between δ-close and ϵ-far inputs, for $\delta < \epsilon$.

Lemma 3.1.1. Let $G = G(M)$ and $L = L(M)$ and $L \cap \{0,1\}^n \neq \emptyset$. Let the automata M be essentially strongly connected as described in Definition 2.2.3. Let m be the reachability constant of $G[C]$. Assume that $\epsilon n \geq 64m \log(\frac{4m}{\epsilon})$. Then, given a word w of length $|w| = n$

1. If w is ϵ-far from L, then, there is some integer $1 \leq i \leq \log(\frac{4m}{\epsilon})$ such that the number of infeasible runs of length 2^{i+1} is at least $\frac{2^{i-4}\epsilon n}{m \log(\frac{4m}{\epsilon})}$ [1]

2. If w is δ-close to L, then for all $1 \leq i \leq \log(\frac{4m}{\epsilon})$ there are at most $2^{i+1}\delta n$ infeasible runs of length 2^{i+1}

Proof outline: We provide a brief outline of the proof. The general claim we are making is there exists a length for which the difference between the number of infeasible runs in words that are δ-close to L and ϵ-far from L is significant. To prove the lemma, we first establish that if w is ϵ-far, then there are many disjoint, infeasible runs. However, when sampling for runs in our algorithm, we must know the size of the runs. Thus, we choose to partition the guaranteed number of infeasible runs by size. By the pigeonhole principle, we see that if a word is ϵ-far from L, then some partition must have many runs of a certain constant size independent of n. The second claim about δ is proven very naturally. We can see that if $\delta < \frac{2^{i-6}\epsilon n}{m\log(\frac{4m}{\epsilon})}$, we will achieve a significant gap in the expected number of runs if a word is δ-close vs. ϵ-far.

This proof can be separated into three parts:

1. Prove that if a word is ϵ-far, then there are many disjoint, infeasible runs

2. In the case where the word is ϵ-far, we show that there exists a size for which there are many infeasible runs. We do this by showing that there are a limited number of size buckets (the "holes" in the pigeonhole principle) that the runs can fit into.

3. We show that if a word is δ-close, there are not that many infeasible runs.

This proof makes use of the concept of minimally infeasible runs defined here

Definition 3.1.1. A run, or continuous sub-word, R of w starting at position i is considered *minimally infeasible* if it is the shortest infeasible run starting at position i. This implies that $R^{(-)}$ obtained by discarding the last bit of R is feasible. This also implies that R' obtained by flipping the last bit of R is feasible.

Proof. The first two parts of the proof are adapted from [1], but reproduced and modified here to fit our algorithm.

We begin by showing that if a word is ϵ-far from being in a language, then there are many disjoint infeasible runs $(R_j)_{j=1}^{h}$. We do so by deliberately constructing these runs to be consecutive and *minimally infeasible* [1] in the sense that all of the prefixes of the run R are feasible, but adding the last bit makes R infeasible. Then, we lower bound the number of minimally infeasible runs by showing a method that can "fix" them all to be feasible.

We only want to find the runs in interval $[m+1, n-m]$ (Recall that m is the connectivity constant from Definition 2.2.1. Our method of "fixing" these infeasible runs relies on being able to modify the m bits before and after the run.). We wish to find the runs in order, so we start at index $m+1$. Let R_1 be the *shortest* infeasible run starting at $w[m+1]$ and ending before $w[n-m+1]$. If there is no such run, we stop. Otherwise, we continue to find the next run starting the index one after where R_1 ended. Let $R_1, ..., R_{j-1}$ end at index c_{j-1}. R_j is the next minimally infeasible run starting at $w[c_{j-1}+1]$ and ending before $w[n-m+1]$. If there is no such run, we stop.

Assume we have constructed h disjoint, minimally infeasible runs $R_1, ..., R_h$ this way. Note that the concatenation of all of these runs forms a continuous subword of w. Additionally, each R_j is minimally infeasible, so its prefix $R_j^{(-)}$ obtained by removing its last bit is feasible. And, so is R'_j, obtained by flipping the last bit in R_j. Using this information, we now provide an method to "fix" w into word $w^* \in L$ and $dist(w, w^*) \leq hm + 2m + 2$. This provides a lower bound on h since $\epsilon n \leq dist(w, w^*)$. The general idea of this method is to fix each subrun individually and "glue" them together to make a feasible word. However, in order to "glue" the subruns together, we may need to change an additional m bits per run, since m is the connectivity constant.

We construct w^* inductively. Our inductive hypothesis is that for each $j = 0, ...$ the constructed word w_j is feasible starting at position 1 and ending at position c_j. For the base case, we let $c_0 = m$ and w_0 be any feasible word of length m starting at position 1. Assume that we have defined w_{j-1} that is feasible from position 1 and ends at c_{j-1}. We will "glue" w_{j-1} with fixed run R'_j obtained by flipping the last bit of minimally infeasible run R_j. Let $\delta(q_1, w_{j-1}) = p_j$. Since R'_j is feasible (Definition [2.2.2]), we know that there is some $q_{i_j} \in C$ such that $\delta(q_{i_j}, R'_j) \in C$ and q_{i_j} is reachable in $c_{j-1}+1$ steps. Then, p_j and q_{i_j} are both reachable in $c_{j-1}+1$ steps from position 1, so we can change the last reachability constant m bits in w_{j-1} to get a word u_j such that $\delta(q_1, u_j) = q_{i_j}$. Now, we can define the feasible subword w_j by concatenating u_j and R'_j.

Since R_h is the last minimally infeasible run, w_h is the final word inductively created and ends at position c_h. Recall that the reason we stopped with R_h is either because 1) there are no more infeasible runs, in which case changing the last m bits of w_h and concatenating it to the remaining suffix of w will achieve $w^* \in L$ or 2) The next minimally infeasible run R_{h+1} starts at index $c_h + 1$ and ends after $n - m + 1$. Then, R'_{h+1} is feasible starting at $c_h + 1$. R'_{h+1} ends within the last m bits of w and R'_{h+1} is feasible, so there must be some word u of length $n - c_h$ where at most m bits are modified such that $\delta(q_{i_h}, u) = q_{acc}$. Then, we construct w^* by concatenating w_h with u the same way we did above.

In the construction of $w^* \in L$, we changed at most m bits per run and modified an additional max $2m + 2$ bits for the ends of the word. Thus $\epsilon n \leq dist(w, w^*) \leq hm + 2m + 2$. Then, the number of minimally infeasible runs $h \geq \frac{\epsilon n - 2}{m} - 2 \geq \frac{\epsilon n}{2m}$, where the last inequality comes from our assumption that $\epsilon n \geq 64m \log(4m/\epsilon)$.

Now, we have a lower bound in the number of minimally infeasible runs. Note that these runs are disjoint. Using this information, we will find a lower bound on the number of runs a certain size. We bucket the runs into $a = \log(4m/\epsilon)$ different size buckets and use the pigeonhole principle to obtain the lower bound.

For $1 \leq i \leq a$, let s_i be the number of disjoint, minimally infeasible runs whose length falls into the interval $[2^{i-1} + 1, 2^i]$. We know that the number of runs of length $\geq \frac{4m}{\epsilon}$ must be $< \frac{\epsilon n}{4m}$ (otherwise, the word would be longer than n bits). By the pigeonhole principle, some size bucket i must have at least $s_i \geq \frac{\epsilon n}{4am}$ disjoint, infeasible runs. We use these s_i runs of variable length to obtain a lower bound on the number of infeasible runs of an exact 2^{i+1} length. Notice that if a run R contains a subrun that is infeasible, R itself is infeasible. Each minimally infeasible run in bucket i is a subword of at least 2^i infeasible runs of length 2^{i+1} (except maybe the first 2 and last 2). Since the minimally infeasible runs are disjoint, at most 3 R_j's fit into one infeasible run of length 2^{i+1}. Thus, there are at least $\frac{2^i}{3}(s_i - 4) \geq \frac{2^i}{3}(\frac{\epsilon n}{4am} - 4) \geq \frac{2^{i-4}\epsilon n}{m\log(4m/\epsilon)}$ infeasible runs of length 2^{i+1}.

Now, we have shown that if a word w is ϵ-far from a language L, for some bucket $1 \leq i \leq \log(4m/\epsilon) = a$, there are at least $\frac{2^{i-4}\epsilon n}{m\log(4m/\epsilon)}$ infeasible runs of length 2^{i+1}. To show a gap between an input word w being δ-close and ϵ-far, we wish to show that for all possible buckets $1 \leq i \leq a$, there are *at most* a certain number of infeasible runs of length 2^{i+1}. To do so, we consider the definition of δ-close. In the worst case scenario, we would have to modify δn bits to obtain a word in L. That means that there are at most δn "bad" bits, or at most δn bits that can make the word infeasible. Every run that does not contain these δn bits ought to be feasible. Let us consider infeasible runs of length 2^{i+1}. Each "bad" bit can participate in at most 2^{i+1} infeasible runs (from the run starting with the bit to the run ending with the bit). Then, since there are δn bad bits, we have at most $2^{i+1}\delta n$ infeasible runs of length 2^{i+1}. This upper bound works across all i and so we have proven our claims. $\square$

In the basic case, where the automata is essentially strongly connected, we have shown there to be a gap between δ and ϵ. Already, a basic tolerant tester becomes apparent in the basic case. For each i, we randomly sample a constant amount (polynomial in $\frac{1}{\delta}$) of runs of length 2^i. Using the upper and lower bounds, we can calculate how many samples we'll need to confidently differentiate between the two cases using the Chernoff or Chebyshev bounds. We can Union bound across the possible buckets to achieve the confidence levels that we need. Now, the question is how we extend the basic case to the general case.

3.2 Extending Beyond the Basic Case

To extend beyond the basic case, we note that each automata M with underlying graph G can be viewed as a group of strongly-connected components. Thus, we define $\mathcal{C}(G)$ to be the *graph of components* whose vertices correspond to the maximal by inclusion strongly connected components of G [1]. To be "maximal by inclusion" means that adding any more nodes of G to the component would violate the strong connectivity property. Some of the vertices of $\mathcal{C}(G)$ may correspond to single vertices of G with no self loops that do not belong to any strongly connected component of G with at least two vertices. We reserve k to be the number of vertices of $\mathcal{C}(G)$ and set $V = V(G)$. We may assume that the underlying DFA M is the minimal DFA [8], which means that all nodes much be reachable from q_1. Then, since $\mathcal{C}(G)$ is consisted of maximal by inclusion connected components, $\mathcal{C}(G)$ is an acyclic graph. Strongly connected component C_1 contains start state q_1 there is a path from C_1 to every other component in the graph.

We make heavy use of a construction used in [1] – *admissible triplets.* An admissible triplet is analogous to an accepting computation history (similar to a computation history used in [15]) of a word w of length n by a DFA M. It describes the high-level path that a word may take through M. Admissible triplets (A, P, Π) consist of three parts: an admissible path of components, a path of portals, and a list of path lengths.

Admissible Path of Components A: We call a path $A = (C_{i_{1,\dots,C_{i_t}}})$ of vertices in $\mathcal{C}(G)$ *admissible* if it starts in component C_1 and ends in a component in an accepting state. The path must traverse along existing edges in $\mathcal{C}(G)$.

Portals P**:** An admissible sequence of portals $P = (p_j^1, p_j^2)_{j=1}^t$ is defined given a length t path of components A. P describes how a word w moves from component to component. The word enters component C_{i_j} at state $p_j^1 \in Q$ (Q is the set of states of the underlying DFA M) and exits component C_{i_j} at p_j^2. Formally, P is defined as such [1]:

1. $p_j^1, p_j^2 \in C_{i_j} \forall 1 \leq j \leq t$

2. $p_1^1 = q_1$

3. $p_t^2 \in F$ (recall F is the set of accepting states of M)

4. For every $2 \leq j \leq t$, $(p_{j-1}^2, p_j^1) \in E(G)$

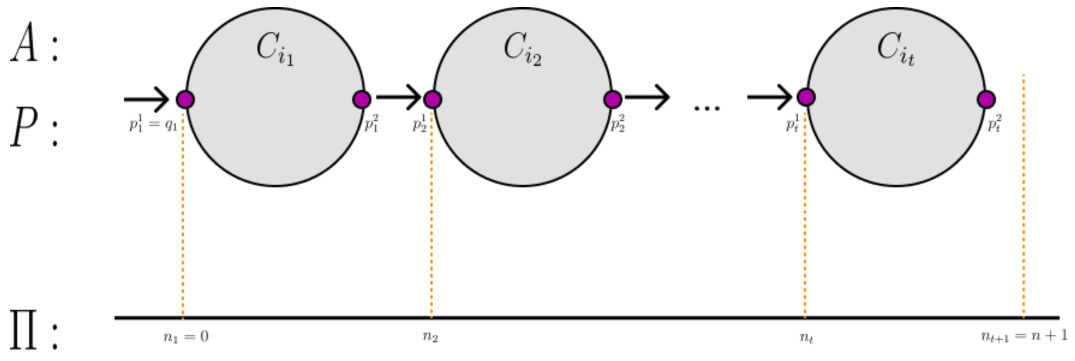

Figure 3-1: An admissible triplet is a generalized possible accepting computation history of an input word w. $A = (C_{i_1}, ..., C_{i_t})$ is the path of strongly connected components that the computation of w goes through. $P = (p_j^1 p_j^2)_{j=1}^t$ (small circles) is entry and exit "portals" of the accepting path through each strongly connected component. Finally, $\Pi = (n_j)_{j=1}^{t+1}$ (dotted lines) corresponds to the the number bits n_j it takes for w to arrive at strongly connected component C_{i_j}. You can calculate the path lengths through each strongly connected component with Π. Note that w always starts at state q_1 in strongly connected component C_1 and must end in an accept state p_j^2.

Admissible Partition Π: Given an admissible path of components A and a corresponding admissible sequence of portals P, we define $\Pi = (n_j)_{j=1}^{t+1}$ to be a sequence of integers that indicates when the word w arrives at component C_{i_j}. Specifically, after n_j bits, the word w arrives at C_{i_j} in in its computation. Increasing sequence of integers $\Pi = (n_j)_{j=1}^{t+1}$ is defined to be an admissible partition if

1. $n_1 = 0$
2. $\forall 1 \leq j \leq t$ there exists a path from p_j^1 to p_j^2 in component C_{i_j} of length $n_{j+1} - n_j - 1$
3. $n_{t+1} = n + 1$

If a word $w \in L$, then clearly there is an admissible triplet associated with w. Therefore, it suffices to check how close or far w is from an admissible triplet to determine whether it is δ-close or ϵ far. We may use the ideas shown in basic case here by testing a word in the connected components indicated by an admissible triplet. We almost have all the tools for the algorithm now, we just need a few more definitions.

3.2.1 Definitions for the General Case

Given an admissible triplet (A, P, Π), we define language L_j corresponding with component C_{i_j} to be all the words accepted by the DFA M_j whose underlying graph is the underlying graph in C_{i_j} union with a garbage state f_j. Specifically, M_j has start state p_j^1 and only one accepting state p_j^2. The set of states in M_j is the set of states in C_{i_j} and its transition function δ_{M_j} is defined as such:

1. For every $q \in C_{i_j}$ and $\sigma \in \{0,1\}$ if $\delta_M(q,\sigma) \in C_{i_j}$, then $\delta_{M_j}(q,\sigma) = \delta_M(q,\sigma)$
2. Else if $\delta_M(q,\sigma) \notin C_{i_j}$, $\delta_{M_j}(q,\sigma) = f_j$
3. $\delta_{M_j}(f_j,\sigma) = f_j$

Then, M_j is essentially strongly connected as defined in Definition 2.2.3. Let L_j be the language of M_j. Given an admissible triplet with a path length of t components, we define subwords $w^1, ..., w^t$ by setting $w_j = w[n_j + 1]...w[n_{j+1} - 1]$ corresponding to the part of w that traverses through component C_{i_j}.

We restate this lemma from [1] without proof.

Lemma 3.2.1. Let (A, P, Π) be an admissible triplet, where $A = (C_{i_1}, ..., C_{i_t})$, $P = (p_j^1, p_j^2)_{j=1}^t$, $\Pi = (n_j)_{j=1}^{t+1}$. Let w be a word of length n where $dist(w, L) \geq \epsilon n$. Let languages $(L_j)_{j=1}^t$ and words $(w_j)_{j=1}^t$ be described above. Then, there exists an index j, $1 \leq j \leq t$ for which $dist(w^j, L_j) \geq \frac{\epsilon n - k}{k}$.

This leads us to the key idea in the proof, we would like to test if w is close to any admissible triplet by sampling runs from w. If not too many infeasible runs are found, we say that w is δ-close. Otherwise, we say that w is ϵ-far.

The only problem is that the number of admissible triplets depends on n, so we can't test each admissible triplet without our algorithm running in at least linear time. This is solved by *transition intervals* [1]. From $1, ..., n$ we place a small number of evenly distributed, constant length transition intervals and say that the transitions between components must happen within these intervals. Since the length of each transition interval will be a function of the maximum connectivity constant m across all components, we can prove that if w is δ-close to L, then only a small modification will allow w to fit an admissible triplet. On the high level, at each transition between strongly connected components, we can use the connectivity constant to modify w to spend a fewer or more steps in any component of our choosing. If w is ϵ-far from L, it will still be far from any admissible triplet.

THIS PAGE INTENTIONALLY LEFT BLANK

Chapter 4

The Algorithm

In this chapter, we provide the tolerant testing algorithm for regular languages and prove its correctness. First, we will provide some preconditions necessary for the algorithm. Then, we will provide the algorithm itself. Finally, we will prove the correctness of the algorithm.

Let connectivity constant $m = \max_j m(C_j)$ and $l = lcm(\{g(G[C_j])\})$ (lowest common multiple of the periods of the connected components). Recall that k is the number of strongly connected components in G, the underlying graph of M. Let $S = 129km\log(\frac{1}{\epsilon})$. We place S transition intervals $T_s = \{[a_s, b_s]\}_{s=1}^{S}$ evenly in $[n]$, where the length of each transition interval T_s is $|T_s| = (k-1)(l+m)$.

In the following algorithm, let p_δ^i be the maximum probability for a single sample of finding an infeasible run of length 2^{i+1} if w is δ-close and let p_ϵ^i be the minimum probability for a single sample of finding an infeasible run of length 2^{i+1} if w is ϵ-far. The probability of finding an infeasible run of length 2^{i+1} with a single uniformly random sample is $\frac{\text{the number of runs of length } 2^{i+1}}{n}$. $p_\epsilon^i = \frac{2^{i-7}\epsilon}{km\log(\frac{1}{\epsilon})}$ for $1 \leq i \leq \log(\frac{8km}{\epsilon})$. With the preconditions below, we can guarantee that $p_\delta^i \leq p_\epsilon/4$ for $1 \leq i \leq \log(\frac{8km}{\epsilon})$.

For $1 \leq i \leq \log(8km/\epsilon)$, we define the number of samples drawn to be $r_i = \frac{64(1-p_\epsilon^i)(2+\log(\frac{1}{\epsilon}^{2k})+\log(\log(\frac{8km}{\epsilon})))}{3(p_\epsilon^i)^2}$

We also place the following preconditions on ϵ, n and δ. This guarantees that n is long enough relative to m and that there is a large enough gap between ϵ and δ.

Preconditions:

1. $\frac{\epsilon n}{2k} \geq 64m\log(\frac{8mk}{\epsilon})$

2. $\epsilon n > 8km$

3. $\epsilon \log(\frac{1}{\epsilon}) < \frac{1}{258k^2|V|^2m(l+m)}$

4. $\delta < \frac{2^{-12}\epsilon}{km\log(\frac{1}{\epsilon})}$

```
Algorithm 1: Tolerant Tester for Regular Languages
  Input: word $w$ of length $n$
  1. for $1 \leq i \leq \log(\frac{8km}{\epsilon})$ do
      1. Choose $r_i$ random runs in $w$ of length $2^{i+1}$
  end
  2. for each admissible triplet $(A, P, \Pi)$, where $A = (C_{i_1}, ..., C_{i_t})$,
     $P = (p_j^1, p_j^2)_{j=1}^t$, $\Pi = (n_j)_{j=1}^{t+1}$ and for all $2 \leq j \leq t$, $n_j \in T_s$ do
      1. Form automata $M_j$, $1 \leq j \leq t$;
      2. Initialize counters for each bucket $1 \leq i \leq \log(\frac{8km}{\epsilon})$
      3. Discard the random runs which end or begin at place $p$ where
         $|p - n_j| \leq \frac{\epsilon n}{128km\log\frac{1}{\epsilon}}$;
      4. For each remaining run $R$, if $R$ falls between $n_j$ and $n_{j+1}$, check to see
         if $R$ is feasible in $M_j$. If $R$ is infeasible, increment the counter for $i$ if
         $|R| = 2^{i+1}$.
      5. If all counters $i$ are less than $Z_i = \frac{p_\delta^i + p_\epsilon^i}{2} \cdot r_i$, this admissible triplet
         matches $w$. Halt the algorithm and output "YES".
  end
  3. If all admissible triplets have too many infeasible runs, output "NO".
```

Before proving the correctness of our algorithm, we would like to provide some information on the process of checking the feasibility of a run R starting at index i. This process takes in constant time relative to the size of the input word w. The main idea is that we make heavy use of Lemma 2.2.1, which specifies the behavior of reachability constant m. First, we use the path lengths defined in admissible partition Π to determine which component C_{i_j} of A we should be be in. We know that component C_{i_j} is either a singular node with no self loops or a essentially strongly connected component. We will focus on the latter case (in the former case, the step

2.3 of the algorithm removes the run from our consideration). Within C_{i_j}, we use associated $n_j \in \Pi$ to discover how many steps past the start state of the associated automata M_j the run R begins its computation. Recall by Definition 2.2.2, we must search across all possible prefixes and suffixes to ascertain if R is feasible. Since the size of automata M_j is constant, this search happens in constant time.

Lemma 4.0.1. If w is ϵ-far from L, then our algorithm outputs "NO" with probability at least $\frac{3}{4}$. If w is δ-close to L, then our algorithm outputs "YES" with probability at least $\frac{3}{4}$.

Proof Outline: The following proof has been split into two parts. First, we will show the case when w is ϵ-far from L, then we will show the case where w is ϵ-close to L. In the case where w is ϵ-far, we show that we don't remove too many runs in line 2.3 of the algorithm so that there is still significant enough of a gap between ϵ and δ that can be tested. In the case where w is δ-close to L, we must show that w matches well enough to at least one of the admissible triplets within the chosen transition intervals. This is technique is adapted from [1] and the main idea is that because the components are strongly connected with connectivity constant m, we can choose to spend more or less time in a connected component to reach the correct transition interval.

We also make significant use of the multiplicative Chernoff bound reproduced here:

$$\mathbb{P}(X \geq (1+\Delta)\mu) \leq e^{\frac{-\Delta^2\mu}{3}}$$

Proof. First, let's consider the case that w is ϵ-far from L. Then, in order for the algorithm to be correct, every single admissible triplet must be found to be infeasible. In order for that to happen, some bucket $1 \leq i \leq \log(8km/\epsilon)$ must have greater than Z_i infeasible runs. First, let's calculate the number of admissible triplets. The number of admissible triplets is bounded by this number [1] [1]:

[1]An admissible triplet is (A, P, Π).We calculate multiply the number of possible paths by the number of possible portals by the number of possible partitions. The number of admissible paths in

$$
\begin{aligned}
& 2^k|V|^{2k}(S(k-1)(l+1))^{k-1} && (4.1)\\
&= 2^k|V|^{2k}(\frac{128km\log\frac{1}{\epsilon}}{\epsilon}(k-1)(l+m))^{k+1} && (4.2)\\
&= 2|V|^2(\frac{2\cdot 129|V|^2km\log(\frac{1}{\epsilon})(k-1)(l+m)}{\epsilon})^{k-1} && (4.3)\\
&< 2|V|^2(\frac{1}{\epsilon}\cdot\frac{1}{\epsilon})^{k-1} && (4.4)\\
&= 2|V|^2(\frac{1}{\epsilon})^{2k-2} && (4.5)\\
&< (\frac{1}{\epsilon})^{2k} && (4.6)
\end{aligned}
$$

Where lines 4.4 and line 4.6 come from the precondition 3.

Now, let's consider how an ϵ-far w behaves in a single admissible triplet (A, P, Π), where $A = (C_{i_1}, ..., C_{i_t})$, $P = (p_j^1, p_j^2)_{j=1}^t$, $\Pi = (n_j)_{j=1}^{t+1}$. Lemma 3.2.1 tells us that there is some component C_{i_j} with corresponding essentially strongly connected automata M_j and the corresponding subword w^j of w is $\frac{\epsilon n}{2k}$ far from $L_j = L(M_j)$. Then, by lemma 3.1.1, for some $1 \leq i \leq \log(\frac{8km}{\epsilon})$, there are at least $\frac{2^{i-6}\epsilon n}{km\log(\frac{1}{\epsilon})}$ infeasible runs of length 2^{i+1}. Since we discard random runs that are too close to the transition points of each component in the admissible triplet, there are at most $2\cdot\frac{\epsilon n}{128km\log(\frac{1}{\epsilon})}$ runs that touch the first and last bits of $[n_j, n_{j+1}-1]$ (corresponding with M_j). Thus, there are at least $\frac{2^{i-7}\epsilon n}{km\log(\frac{1}{\epsilon})}$ infeasible runs of length 2^{i+1}. Then, as defined above for some i $p_\epsilon^i = \frac{2^{i-7}\epsilon}{km\log(\frac{1}{\epsilon})}$.

We model each sample x_a^i of length 2^{i+1} as an indicator random variable with probability p_ϵ^i. Then, let $X_i = \sum_{a=1}^{r_i} x_a^i$. The probability for each i that we believe the word w to be in L when it is not is $\mathbb{P}(X_i \leq Z_i)$, where $Z_i = \frac{p_\epsilon^i + p_\delta^i}{2}\cdot r_i$. We bound

$\mathcal{C}$ is at most 2^k. The number of possible portals is at most $|V|^{2k}$, then for each (A, P), the number of admissible paths is at most $S|T_s|$ for each n_j $2 \leq j \leq t$, $t \leq k-1$).

this with the Chernoff bound. Let $Y_i = r_i - X_i$

$$\begin{aligned}
\mathbb{P}(X_i \leq Z_i) &= \mathbb{P}(Y_i \geq r_i - Z_i) && (4.7)\\
&\leq \mathbb{P}(Y_i \geq r_i(1 - \frac{5}{8})r_i) && (4.8)\\
&= \mathbb{P}(Y_i \geq (1+\Delta)\mathbb{E}(Y_i)) && (4.9)\\
&= \mathbb{P}(Y_i \geq (1+\Delta)(1 - p_\epsilon^i r_i)) && (4.10)\\
&\leq e^{-\frac{\Delta^2(1-p_\epsilon^i)r_i}{3}} && (4.11)\\
&\leq e^{-(2+\log(\log(8km/\epsilon))+\log(\frac{1}{\epsilon}^{2k}))} && (4.12)\\
&= e^{-2}\frac{1}{\log(8km/\epsilon)}\frac{1}{\epsilon^{2k}} && (4.13)
\end{aligned}$$

Where $\Delta = \frac{1-\frac{5}{8}p_\epsilon^i}{1-p_\epsilon^i}$ and line 4.8 comes from Precondition 4. We use the Union Bound across all buckets $1 \leq i \leq \log\frac{8km}{\epsilon}$ and then the Union Bound across all triplets. Then, we achieve that the probability of our algorithm failing when w is ϵ far is at most $e^{-2} \leq \frac{1}{4}$

Now, we consider the case where w is δ-close to L. First, we will show that if some w^* is in L, then there is an admissible triplet that passes successfully. We need to show that although we are limiting the triplets with transition intervals, there is still some triplet that is close enough that will pass w^* in the process described by our algorithm. After showing that there exists such a triplet, we will show that if w is δ-close to L, w will be δ close to the admissible triplet in question. In particular, we will show for that admissible triplet, there is a low probability that more than Z_i infeasible runs will be found for all buckets i.

First let's consider w^* which we will say is the closest word to w in L. Then, we know $dist(w, w^*) \leq \delta n$. Since w^* is in L, then we know there is a path that w^* traverses through finite automata M which naturally defines an admissible triplet (A, P, Π). $A = (C_{i_1}, ..., C_{i_t})$ which is the components through $\mathcal{C}(G)$ in the order that w^* traversed them. $P = (p_j^1, p_j^2)_{j=1}^t$, where p_j^1 and p_j^2 are respectively the first and last states that w visited in C_{i_j}. Finally, $\Pi = (n_j)_{j=1}^{t+1}$, where $n_1 = 0$ and $n_{t+1} = n+1$. For all other j, n_j is represents the first time w^* enters component C_{i_j}. However, the issue here is that the admissible triplet which the traversal of w^* defines may not have transitions in the transition intervals. We will show that an (A, P, Π) associated with $w^* \in L$ can be modified in Π only so that the modified triplet (A, P, Π') is associated with another $w' \in L$ (very close to w^*) that follows the same path of components as w^* [1]. Then, as long as we make sure that no query is made to bits of w' that differ from w^* (Step 2.3 of the algorithm discards runs that are too close to the transition intervals), w should look δ close to w' as well.

Let's consider (A, P, Π) as defined above for w^*. Let's say for some $1 \leq j \leq t$, n_j is not in any of the randomly selected transition intervals. Intuitively, using the connectivity constant m of the automata M, we can make it so that we either spend fewer or more steps inside component $C_{i_{j-1}}$ so that we exit the component within transition interval [1] (which is $(k-1)(l+m)$ long to guarantee our ability to do this). We do this as follows: $\Pi' = (n'_j)_{j=1}^{t+1}$, where $n'_1 = n_1 = 0$ and $n'_{t+1} = n_{t+1} = n+1$. For every other j, we choose transition interval T_s closest to n_j. If the underlying M_j for C_{i_j} is essentially strongly connected, we choose n'_j such that a) $n'_j \equiv n_j \mod l$ (recall l is the lcm of all the periods of the M_j's) and b) $n'_j - n'_{j-1} > m$. If C_{i_j} is a singleton without loops, we set $n'_j = n'_{j-1} + 1$. As $|T_s| = (k-1)(l+m)$, we know n'_j always exists. Thus, we obtain modified triplet (A, P, Π') [1].

We claim (A, P, Π') is admissible [1]. For all $1 \leq j \leq t$, we have $n'_{j+1} - n'_j \equiv n_{j+1} - n_j \mod l \implies n'_{j+1} - n'_j \equiv n_{j+1} - n_j \mod g(G[C_{i_j}])$ if M_j is essentially strongly connected. Then, by Lemma 2.2.3, if there exists a path of length $n_{j+1} - n_j - 1$ through M_j, there exists a path of length $n'_{j+1} - n'_j - 1$. This implies that if (A, P, Π) is admissible, then (A, P, Π') is admissible.

Now we show that our procedure in step 2.3 of the algorithm only queries bits of w^* that are identical to bits of w'. We want to show that our algorithm does not query the differences between w^* and w' [1]. Let R be a run that is $\frac{\epsilon n}{128km \log \frac{1}{\epsilon}}$ close to a transition interval. In particular, $R \in [n'_j + \frac{\epsilon n}{128km \log \frac{1}{\epsilon}}, n'_{j+1} - \frac{\epsilon n}{128km \log \frac{1}{\epsilon}}]$. Let b be the first index of R. Since we placed the S transition intervals evenly apart in $[n]$, we have $|n'_j - n_j| \leq \frac{n}{S} + |T_s| = \frac{\epsilon n}{128km \log \frac{1}{\epsilon}} + (k-1)(l+m)$. Then, R falls completely in $[n_j + m, n_{j+1} - 1]$ and R is feasible for M_j. Therefore, we delete only the runs that have one of the ends too close to the transition interval because it may be that R starts in a place where w^* is in $C_{i_{j-1}}$ and ends in a place where w^* is in C_{i_j}. Thus, discarding the marginal runs too close to the transition intervals will ensure that there is no query difference in the algorithm between w^* and w' on triplet (A, P, Π'). Then, lemma 2.2.3 guarantees that if we were sampling from w^*, all runs sampled (except those removed) would be feasible in (A, P, Π) [1].

Since our input w is δ-close to w^*, there are at most δn infeasible runs that have ends that are not too close to the transition intervals (i.e. we don't remove these infeasible runs from consideration in step 2.3 of the algorithm). We show that for the correct admissible triplet (A, P, Π'), we do not find more than Z_i infeasible runs for each bucket $1 \leq i \leq \log \frac{8km}{\epsilon}$. Once again, let X_i be the sum of the indicator variables x_a^i for each sample a. x_a^i is 1 if an infeasible run is found. Thus, x_a^i is 1 with probability p_δ.

$$\begin{aligned}
\mathbb{P}(X_i \geq Z_i) &= \mathbb{P}(X_i \geq \frac{p_\delta + p_\epsilon}{2} r_i) && (4.14)\\
&= \mathbb{P}(X_i \geq \frac{p_\epsilon 4 + p_\epsilon}{2} r_i) && (4.15)\\
&= \mathbb{P}(X_i \geq \frac{5}{8} p_\epsilon r_i) && (4.16)\\
&= \mathbb{P}(X_i \geq (1+\Delta) r_i \frac{p_\epsilon}{4}) && (4.17)\\
&\leq e^{-((\frac{3}{2})^2 r_i p_\epsilon / 4/3)} && (4.18)\\
&= e^{-\frac{3}{16} r_i p_\epsilon} && (4.19)\\
&= e^{-(4 \cdot \frac{1}{p_\epsilon}(1-p_\epsilon)(2+\log(\frac{1}{\epsilon}^{2k} + \log\log\frac{8km}{\epsilon})))} && (4.20)\\
&\leq e^{-4(2+\log\log\frac{8km}{\epsilon})} && (4.21)\\
&\leq e^{-8} \frac{1}{\log \frac{8km}{\epsilon}} && (4.22)
\end{aligned}$$

Where Δ from line 4.17 is $\frac{3}{2}$. Union Bound across all i, the probability of failure is at most $e^{-8} \leq \frac{1}{4}$. Thus if w is δ-close to L, we say "YES" with probability at least $\frac{3}{4}$.

□

Now, we consider the query complexity of the algorithm. There are $\sum_{i=1}^{\log \frac{8km}{\epsilon}} 2^{i+1} r_i$ queries, which is polynomial in $\frac{1}{\epsilon}$.

Thus, we have proven the theorem below.

Theorem 4.0.2. For every regular language L, every integer n, every small enough $\epsilon > 0$ and $\delta > 0$, there exists a tolerant testing algorithm with query complexity polynomial in $\frac{1}{\epsilon}$ for $L \cup \{0,1\}^n$.

Note that in our preconditions, our algorithm works for δ bounded by Precondition 4. Our method would actually work for any δ, as long as $p_\delta < p_\epsilon$; however, r_i would be chosen to be a different number (the query complexity). We chose δ such that $p_\delta = p_\epsilon/4$ for simplicity of analysis.

Chapter 5

Conclusion and Future Work

The main technical achievement of our thesis provides the first tolerant tester for regular languages. We believe that an interesting direction to look into is a local computation algorithm (LCA) for fixing the input word w so that it belongs in regular language L [14] [2]. A local computation algorithm would support user queries to a set of locations in input word w, responding with the fixed values of w in those locations. If the sample of w was an infeasible run, the values returned by the LCA would be feasible runs. The local computation algorithm must "fix" the query to w in a manner such that it is consistent regardless of the past and future queries.

It may also be interesting to consider testing the class of context-free languages (CFLs), which are languages recognized by pushdown automata [15] and context-free grammars. To our knowledge, there are no known testers, tolerant or otherwise, for CFLs. However, we do know that there exists a lower bound of $\Omega(\sqrt{n})$ to test CFLs [1]. Since tolerant testing is usually a harder question than nontolerant testing, we believe more reasonable open questions involve the tolerant testing Dyck languages [3] or the tolerant testing of parenthesis languages [11]. We could also consider tolerant testing of specific CFLs, for example those of the form uu^Rvv^R [1].

THIS PAGE INTENTIONALLY LEFT BLANK

Bibliography

[1] Noga Alon, Michael Krivelevich, Ilan Newman, and Mario Szegedy. Regular languages are testable with a constant number of queries. *Foundations of Computer Science, 1975., 16th Annual Symposium on*, 30, 05 2001.

[2] Noga Alon, Ronitt Rubinfeld, Shai Vardi, and Ning Xie. Space-efficient local computation algorithms, 2011.

[3] Eldar Fischer, Frédéric Magniez, and Tatiana Starikovskaya. Improved bounds for testing dyck languages, 2017.

[4] Peter Gemmell, Richard Lipton, Ronitt Rubinfeld, Madhu Sudan, and Avi Wigderson. Self-testing/correcting for polynomials and for approximate functions. In *Proceedings of the Twenty-Third Annual ACM Symposium on Theory of Computing*, STOC '91, page 33–42, New York, NY, USA, 1991. Association for Computing Machinery.

[5] Oded Goldreich, Shafi Goldwasser, Eric Lehman, Dana Ron, and Alex Samorodnitsky. Testing monotonicity. *Combinatorica*, 20(3):301–337, 2000.

[6] Oded Goldreich, Shari Goldwasser, and Dana Ron. Property testing and its connection to learning and approximation. *J. ACM*, 45(4):653–750, July 1998.

[7] Oded Goldreich, Tom Gur, and Ron D. Rothblum. Proofs of proximity for context-free languages and read-once branching programs. *Information and Computation*, 261:175–201, 2018. ICALP 2015.

[8] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation (3rd Edition)*. Addison-Wesley Longman Publishing Co., Inc., USA, 2006.

[9] Mordechai Lewin. A bound for a solution of a linear diophantine problem. *Journal of the London Math Society*, 1972.

[10] Antoine Ndione, Aurélien Lemay, and Joachim Niehren. Approximate membership for regular languages modulo the edit distance. *Theoretical Computer Science*, 487:37–40, 2013.

[11] Michal Parnas, Dana Ron, and Ronitt Rubinfeld. Testing membership in parenthesis languages. *Random Struct. Algorithms*, 22(1):98–138, January 2003.

[12] Michal Parnas, Dana Ron, and Ronitt Rubinfeld. Tolerant property testing and distance approximation. *Journal of Computer and System Sciences*, 72(6):1012–1042, 2006.

[13] Ronitt Rubinfeld and Madhu Sudan. Robust characterizations of polynomials withapplications to program testing. *SIAM J. Comput.*, 25(2):252–271, February 1996.

[14] Ronitt Rubinfeld, Gil Tamir, Shai Vardi, and Ning Xie. Fast local computation algorithms, 2011.

[15] Michael Sipser. *Introduction to the Theory of Computation.* International Thomson Publishing, 3rd edition, 2013.