NOVA SCHOOL OF SCIENCE & TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE

GONÇALO FILIPE RASTEIRO CAVACO

BSc in Computer Science and Engineering

LEARNING GENES FROM MUTATED COPIES: ALGORITHMS AND COMPLEXITY

MASTER IN COMPUTER SCIENCE AND ENGINEERING

NOVA University Lisbon
July, 2024

NOVA SCHOOL OF SCIENCE & TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE

LEARNING GENES FROM MUTATED COPIES: ALGORITHMS AND COMPLEXITY

GONÇALO FILIPE RASTEIRO CAVACO

BSc in Computer Science and Engineering

Adviser: João Miguel Lourenço Ribeiro
Assistant Professor, NOVA University Lisbon

Examination Committee

Chairs: Ricardo João Rodrigues Gonçalves
Assistant Professor, NOVA University Lisbon

André Nuno Carvalho Souto
Assistant Professor, University Lisbon

MASTER IN COMPUTER SCIENCE AND ENGINEERING

NOVA University Lisbon
July, 2024

Learning genes from mutated copies: Algorithms and complexity

This document was created with the (pdf/Xe/Lua)LaTeX processor and the NOVAthesis template (v6.10.18) [15].

Acknowledgements

First and foremost, I would like to thank my advisor João Ribeiro for his incredible help and advice. His guidance was crucial to overcome some challenges of this work, namely the theoretical aspects of it. I want to thank him also for allowing me to implement my ideas, which have contributed positively to this work.

I want to thank the Department of Computer Science of NOVA-FCT, for the last five years. I believe, with all the knowledge and guidance I have received, I'm ready for what lies ahead.

A special thanks to all the friends I have made in the last few years. Their support and companionship were extremely important to help me be better every day.

I would like also to thank my parents for always supporting me and allowing me to pursue what I enjoy the most. Finally, I want to thank a very special person that, without him, none of this could be possible. I'm eternally grateful to my Mathematics High School teacher, Joaquim Caetano, for advising me to pursue this field of study. He was right that Computer Science would be the perfect career for me.

”

“Valeu a pena? Tudo vale a pena, se a alma não é pequena.”

— **Fernando Pessoa**, Mensagem, Mar Português (pt)

Abstract

In our world, data is everywhere. It is transmitted between devices, or simply stored in the disc. In these two processes, data can be corrupted, simply for a change, erasure, or even deletion in its content. The generated corrupted sequences are what we call traces and, sometimes, we only have access to them, leaving the original content unknown. This is why we study trace reconstruction algorithms, to try to recover the original content from the corrupted traces, using the minimum traces possible, with high probability. The problems we will approach are over DNA sequences of biological models.

The main challenge in trace reconstruction is to determine the minimum number of necessary traces to achieve accurate reconstruction within the upper bounds of $O(n \log(n))$ traces and lower bounds of $\Omega(n \log(n))$ traces. In this thesis, we propose the study of trace reconstruction over DNA biological models. We will study two different settings, the average-case and worst-case reconstruction and use reconstruction techniques like mean-based reconstruction. When the analysis is hard or we believe we can achieve better results, we will resort to genetic algorithms to get better upper bounds.

From this work, we expect to derive rigorous theorems over the sample complexity, i.e., the optimal number of traces needed for reliable reconstruction of the problems of trace reconstruction, motivated by the biological models. When the analysis of the models proves to be very difficult, we will resort to empirical arguments with genetic algorithms.

Keywords: trace reconstruction, genetic algorithms, DNA sequences, Upper bounds, Lower Bounds

Resumo

No nosso mundo, os dados estão em todo o lado. Eles são transmitidos entre dispositivos, ou simplesmente guardados em disco. Nestes dois processos, os dados podem ser corrompidos, simplemente por uma mudança, um apagamento, ou uma eliminação do seu conteúdo. As sequências corrompidas geradas são o que chamamos de traços e, às vezes, somente temos acesso a eles, permanecendo o conteúdo original desconhecido. É por isto que são estudados algoritmos de *trace reconstruction* com o propósito de, através dos traços corrompidos, recuperar o conteúdo original, usando o mínimo de traços possível, com alta probabilidade. Os problemas que iremos abordar, são sobre modelos biológicos, sobre sequências de DNA.

O principal desafio em *trace reconstruction* é determinar o número mínimo de traços necessários para alcançar uma reconstrução precisa, para majorantes na ordem de $O(n \log(n))$ e minorantes de $\Omega(n \log(n))$.

Nesta tese, nós propomos o estudo de problemas *trace reconstruction* sobre problemas de modelos biológicos de DNA. Nós iremos estudar dois scenários diferentes, o *average-case reconstruction* e *worst-case reconstruction* e usar estratégias como *mean-based reconstruction*. Quando a análise se demonstrar difícil ou acreditamos que conseguimos melhores resultados, utliaremos algoritmos genéticos.

Deste trabalho, nós esperamos derivar teoremas rigorosos sobre a complexidade amostral i.e., o número ótimo de traços necessários para uma reconstrução fiável, sobre os problemas de *trace reconstruction*, motivados por modelos biológicos. Quando esta análise se demonstrar muito difícil,usaremos argumentos empíricos, utilizando redes neuronais.

Palavras-chave: *trace reconstruction*, *algoritmo genético*, sequências de DNA, *Upper bounds*, *Lower bounds*

Contents

List of Figures

List of Listings

1 Introduction

1.1 Motivation

In today's data-driven world, we rely on data storage and transmission more than ever. However, during these processes, data can be subjected to errors, that compromise its integrity. So it is important to encode data, to recover it from errors.

The mutation, erasure and deletion of symbols are among the most common errors. Suppose we want to transmit a message to a person and the receptor receives, with some probability, the message with the symbol "?". If we re-transmit this message a few times we will have several corrupted sequences and here is when Trace Reconstruction comes in and is important. In trace reconstruction, we want to design algorithms that, based on i.i.d (independent and identically distributed) corrupted sequences that we call traces, recover the original unknown content with high probability. Real-world illustrations include the magnetic and digital storage [12, 20] and biological models based on the mutated copies of DNA sequences [3]. We will explore trace reconstruction algorithms, which are algorithms, based on the corrupted content of a string that we call traces, and try to estimate efficiently and with high probability the input string. Beyond this, we want to define the best sample complexity over these algorithms, i.e., upper bounds in the order of $O(n \log(n))$ traces and lower bounds of $\Omega(n \log(n))$ traces.

Our main motivation relies on the field of trace reconstruction over the biological models described in [3] because they provide valuable insights and inspiration for algorithmic development and no one has already analysed them. Biological systems, such as genetic sequences of DNA, exhibit complex dynamics that can be studied and modelled to enhance the reconstruction process. By exploring these biological models, we can gain a deeper understanding of the underlying principles governing trace reconstruction and apply this knowledge to develop more efficient algorithms. In this thesis, our models will be binary strings of arbitrary length n, i.e., strings over the alphabet $\{0,1\}^n$.

1.2 Problem Statement and Challenges

Bhardwaj et al. [3] have proposed many models that have not been explored yet. We found these models interesting and challenging to explore and analyse, so we propose to explore trace reconstruction over them.

As we already said, trace reconstruction involves reconstructing the unknown input content based on its traces. But, determining the best sample complexity is the most challenging aspect of this. So, for these models, we want to reach upper bounds of $O(n \log(n))$ traces and lower bounds of $\Omega(n \log(n))$ traces.

There exist two different reconstruction settings: the worst-case and average-case reconstruction. The worst-case analysis aims to provide an algorithm that efficiently reconstructs each symbol, for every arbitrary input string with high probability [16]. The average-case analysis considers algorithms that are capable of reconstructing, with high probability, on average over all input strings[16]. One of the most common techniques for worst-case trace reconstruction problems is the mean-based reconstruction algorithm. This technique leverages the statistical properties of the traces, namely, the expected value coordinate by coordinate, over all traces, to estimate the original sequences, taking into account the distributional characteristics of the data [7]. With these settings, we can study the sample complexity i.e., the optimal number of input samples needed to make an accurate reconstruction. Additionally, by studying the application of these algorithms in biological models, we can assess their strengths, limitations, and suitability for different scenarios. Additionally, beyond the formal analysis of the algorithms, we want to implement them, so we can have empirical proof of the theorems and estimate the average-case reconstruction error. In parallel, we will develop a genetic algorithm to do some experiments to find out if we can achieve better results, in terms of sample complexity.

1.3 Objectives

In the most general way, our goal with this work is to study Trace Reconstruction Concretely, we want to

- Study the sample complexity of various trace reconstruction algorithms over the models proposed in [3].
- Develop highly efficient reconstruction algorithms with nearly optimal sample complexity.
- Implement reconstruction algorithms in Python to empirically estimate the average-case error.
- Build a genetic algorithm to try to improve the sample complexity results.

1.4 Document Structure

This document is structured in the following way:

- Chapter 1, *Introduction*, introduces the context and motivation for this thesis, the problem statement and the corresponding challenges, objectives and expected contributions.
- Chapter 2, *Background*, provides the mathematical concepts which are the base for this thesis.
- Chapter 3, *State of the Art*, provides the state-of-the-art algorithms, their upper and lower bounds, some specific definitions and the proposed work.
- Chapter 4,*TrimSuffixAndExtendModel*, rigorous analysis of the trimSuffixAndExtend model, including Lower and Upper bounds.
- Chapter 5 *TrimSuffixAndExtend with Mutations*, beginning of the analysis of upper bounds of the model.
- Chapter 6 *Experimental results*, Empirical results, given by the implementation of the developed algorithms and the genetic algorithm.
- Chapter 7 *Conclusions and Future Work*, presents the conclusions of the research work as well as the future directions

2

Background

In this chapter, we will describe all the fundamental concepts that will be essential to understanding how noise models work and their properties. Several concepts from probability theory, such as probability axioms, the definition of discrete random variables, their properties and important probability distributions will be described. Also, some important concepts like Markov and Chernoff bounds will be defined. Finally, we will introduce the information-theoretic concepts, their properties, and some basics of boolean function analysis.

2.1 Probability Theory

In this section, we will describe all the probability theory background needed to understand the models that are forthcoming in Chapter 3.

2.1.1 Set theory

An event E represents the set of all possible outcomes of a probabilistic experiment. We use standard set theory notation to express combinations of events [17]. To express the intersection of events we write, $E_1 \cap E_2$, where both events must co-occur. To represent their union, we use $E_1 \cup E_2$, where one of them or both can happen. Ultimately, we denote by $\bar{E}$ the event $\Omega - E$, meaning all the probability space except E [17].

2.1.2 Important Lemmas and Theorems

Theorem 2.1.1 (Law of Total Probability)**.** *Let $E_1, E_2, \ldots, E_n$ be mutually disjoint events in the sample space Ω, and let $\bigcup_{i=1}^{n} E_i = \Omega$. Then*

$$\Pr(B) = \sum_{i=1}^{n} \Pr(B \cap E_i) = \sum_{i=1}^{n} \Pr(B|E_i) \Pr(E_i).$$

Theorem 2.1.2 (Bayes' theorem). *Assuming that E_1 , E_2 , ... , E_n are mutually disjoint sets such that $\bigcap_{i=1}^{n} E_n = E$. Then*

$$\Pr(E_j|B) = \frac{\Pr(E_j \cap B)}{\Pr(B)} = \frac{\Pr(B|E_j)\Pr(E_j)}{\sum_{i=1}^{n} \Pr(B|E_j)\Pr(E_j)}.$$

2.1.3 Discrete Random Variables and Expectation

Definition 2.1.1 (Expectation of a random variable Definition 2.3 [17]). *The expectation of a random variable is the "weighted average of the values it assumes, where each value is weighted by the probability that the variable assumes that value". Mathematically, we define it as*

$$E[X] = \sum_{x} x \Pr(X = x).$$

Theorem 2.1.3 (Linearity of Expectations Theorem 2.1 [17]). *For every finite collection of discrete random variables $X_1, X_2, \ldots X_n$ with finite expectations*

$$E\left[\sum_{i=1}^{n} X_i\right] = \sum_{i=1}^{n} E[X_i].$$

Definition 2.1.2 (Bernoulli random variable Section 2.2 [17]). *A variable X is a Bernoulli random variable, with a success probability of p if*

$$X = \begin{cases} 1, & \textit{with probability } p \\ 0, & \textit{with probability } 1-p \end{cases}$$

The expected value of X is calculated as

$$E[X] = 1 \cdot p + 0 \cdot (1-p) = p = \Pr(X = 1).$$

If we consider now n independent Bernoulli events and j number of successes in the n experiments, we are in the presence of a Binomial distribution.

Definition 2.1.3 (Binomial distribution Definition 2.5 [17]). *X has Binomial distribution of parameters (n, p) is defined as*

$$X = \sum_{i=1}^{n} Z_i,$$

where Z_i are i.i.d. Bernoulli random variables

The expected value of a Binomial value is

$$E[X] = np.$$

Definition 2.1.4 (Geometric random variables Definition 2.8 [17]). *A geometric random variable X with parameter p is described as*

$$\Pr(X = n) = (1-p)^{n-1}p.$$

This means that "for the geometric random variable X to equal n, there must be $n-1$ failures"[17].

2.1.4 Moments and Deviations of Random Variables

Definition 2.1.5 (Moment of a random variable Definition 3.1 [17]). *The k-th moment of X is defined to be* $E[X^k]$

Definition 2.1.6 (Variance of a random variable Definition 3.2 [17]). *The variance of a random variable can be defined as*

$$Var[X] = E[(X - E[X])^2] = E[X^2] - (E[X])^2.$$

With the variance, the standard deviation can be computed as

$$\sigma[X] = \sqrt{Var[X]}.$$

2.1.5 Important Inequalities

Theorem 2.1.4 (Markov's inequality Theorem 3.1[17]). *Being X a random variable that assumes only nonnegative values and, for all* $a > 0$*, we have*

$$\Pr(X \geq a) \leq \frac{E[X]}{a}$$

Definition 2.1.7 (Moment Generating Function Definition 4.1 [17]). *The moment-generating function of a random variable X captures all moments of X. It is defined as*

$$M_X(t) = E[e^{tX}].$$

The main interest in this is the fact that, around zero, the n^{th} moment of X, is defined as

$$E[X^n] = M_X^{(n)}(0),$$

where $M_X^{(n)}(0)$ is the n^{th} derivative of $M_X(t)$ when $t = 0$ [17]. Chernoff Bounds are derived by applying Markov's inequality to e^{tX} for a chosen t [17]. So, for any $t > 0$, we have that

$$\Pr(X \geq a) = \Pr(e^{tX} \geq e^{ta}), \tag{2.1}$$

which by application of Theorem 2.1.4 in eq. (2.1), we have

$$\Pr(e^{tX} \geq e^{ta}) \leq \min_{t<0} \frac{E[e^{tX}]}{e^{ta}}.$$

Chernoff bounds are mainly applied to the upper bound on the left, right or both tails of distributions [17]. Consider $X = \sum_{i=1}^{n} X_i$, where X_i is a 0-1 random variable and $\mu = E[X]$. Considering this, we can write the following Chernoff Bounds.

Theorem 2.1.5 (Right tail, left tail and double tail Chernoff bounds Theorem 4.4, 4.5 and Corollary 4.6 [17])**.**

$$\Pr(X \geq (1+\delta)\mu) \leq e^{-\mu\delta^2/3}, 0 < \delta \leq 1 \tag{2.2}$$

$$\Pr(X \leq (1-\delta)\mu) \leq e^{-\mu\delta^2/2}, \ 0 < \delta < 1., \tag{2.3}$$

$$\Pr(|X - \mu| \geq \delta\mu) \leq 2e^{-\mu\delta^2/3}, 0 < \delta < 1. \tag{2.4}$$

3

State of the Art

In this Chapter, we will describe and analyse three virtual channels in the area of trace reconstruction, in particular, the Binary Symmetric Channel, the Binary Erasure Channel and the Binary deletion Channel). We will also present some prior work on the application of these channels.

3.1 Important Definitions

Before proceeding into the problems, we will define some important concepts to help us better understand the models.

3.1.1 Alphabet

Definition 3.1.1 (Alphabet [1])**.** *We denote by an alphabet a discrete non-empty set $\mathcal{X}$, composed of elements, named symbols.*

From them, we can construct sequences of symbols that we denote by strings.

Definition 3.1.2 (String)**.** *A string x is a finite sequence of symbols over an alphabet.*

The set $\mathcal{X}^*$ denotes all the finite sequences of elements of $\mathcal{X}$. Mostly, in this work, we will work over a binary alphabet $\{0, 1\}$.

3.1.2 Channel

Definition 3.1.3 (Channel Definition 2.3, [16])**.** *A channel, Ch, in its most general way, is described as something that has as input an alphabet $\mathcal{X}$ and output an alphabet $\mathcal{Y}$. It receives, as input, a string $x \in \mathcal{X}^*$ and outputs random variable Y_x supported on $\mathcal{Y}^*$.*

In our case, we are mostly interested in discrete memoryless channels (DMCs), that is, channels that receive an ordered input and do not keep the state of the processed elements [16]. These types of channels process information like this: For each $x_i \in \mathcal{X}$ that belongs

to a string $x_1, \ldots, x_n$, the channel Ch outputs a discrete random variable Y_x supported on $\mathcal{Y}^*$, resulting in

$$Y_x = Y_{x_1} || Y_{x_2} || \ldots || Y_{x_n}$$

where $||$ means string concatenation and $Y_{x_1} || Y_{x_2} || \ldots || Y_{x_n}$ are independent according to the channel output distribution.

DMCs map each single input symbol into a single output symbol, satisfying, for $x \in \mathcal{X}^n$

$$Y_x(y) = \prod_{i=1}^{n} Y_{x_i}(y_i),$$

where $Y_{x_i}(y_i)$ is the probability of the channel outputs y_i on input x_i.

Definition 3.1.4 (Hamming Weight [21])**.** *Given a string $x \in \mathcal{X}^n$, we denote the Hamming Weight of x by $wgt(x)$, which is defined as*

$$wgt(x) = \#\{i \in \{1, \ldots, n\} \mid x_i \neq 0\},$$

. where # denotes the cardinality of a set.

Next, we will define a tool to measure how two given strings x, y differ in their characters. This is called the Hamming Distance.

Definition 3.1.5 (Hamming Distance [21])**.** *The Hamming Distance between two strings $x, y \in \{0,1\}^n$ is defined as*

$$\Delta(x, y) = \#\{i : x_i \neq y_i\}$$

.

Definition 3.1.6 (L1 Norm [8])**.** *The ℓ_1 norm between two strings can be defined as*

$$\|x - y\|_1 = \sum_{i=0}^{n} |x_i - y_i|, \tag{3.1}$$

where $|x_i - y_i|$ denotes the absolute value between $x_i - y_i$.

3.2 Trace Reconstruction

Suppose we have a fixed string x, of symbols in an alphabet $\mathcal{X}$ that passes through a channel Ch, t times, generating t traces. Each trace is generated by modifying the bits from the input string and applying mutations, erasures or deletions over them, depending on the channel output alphabet. Let x_i be the i^{th} bit of x and each $x_i \in \{0,1\}$. Each x_i is turned into a y_i according to the channel output alphabet $\mathcal{Y}^n$. More precisely, a sequence x, $(x_1, \ldots, x_n)$, is transformed in t traces $(y^{(1)}, \ldots, y^{(t)})$, being each $y^{(t)}$ i.i.d. Keeping this in mind, we want to

- Estimate the sample complexity for the upper and lower bound to reach, respectively, $O(\log(n))$ and $\Omega(\log(n))$

- Design algorithms to estimate the input string efficiently.

We will describe and analyse three important channels:

- **Binary Symmetric Channel**: where each bit is flipped with probability ε.

- **Binary Erasure Channel**: where each bit is erased with probability ε.

- **Binary Deletion Channel**: where each bit is deleted with probability ε

Given this, we will noe explore each one of these channels separately.

3.2.1 Binary Symmetric Channel

The Binary Symmetric Channel (BSC) receives, as input, bits supported in $\mathcal{X}$ and flips each bit independently with probability ε. So, we can define the input and the output alphabets as $\mathcal{X} = \mathcal{Y} = \{0, 1\}$. Let $x_i \to y_i$ denote the transformation of the bit x_i into y_i when it passes through the channel. Given this, we can write

$$x_i \to y_i = \begin{cases} x_i, & \text{with probability } 1 - \varepsilon \\ 1 - x_i, & \text{with probability } \varepsilon. \end{cases}$$

Fig. 3.1 illustrates how this model works.

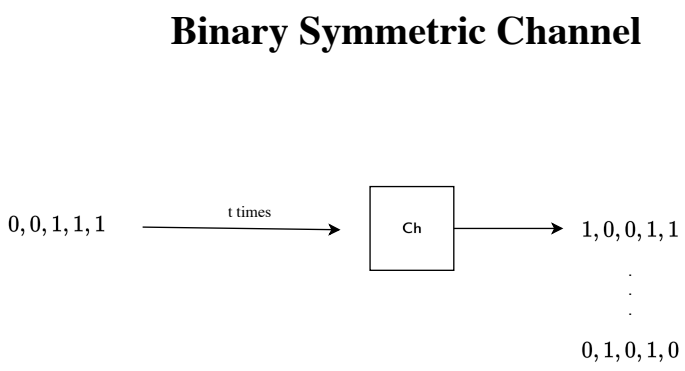


Figure 3.1: Binary Symmetric Channel

Another example of a DMC is the Binary Erasure Channel.

3.2.2 Binary Erasure Channel

The Binary Erasure Channel (BEC) receives a bit string as input and independently erases each bit with probability ε. An example of a binary erasure channel can be characterised by Fig. 3.2 In this type of channel, the input alphabet $\mathcal{X} = \{0, 1\}$ and the output alphabet

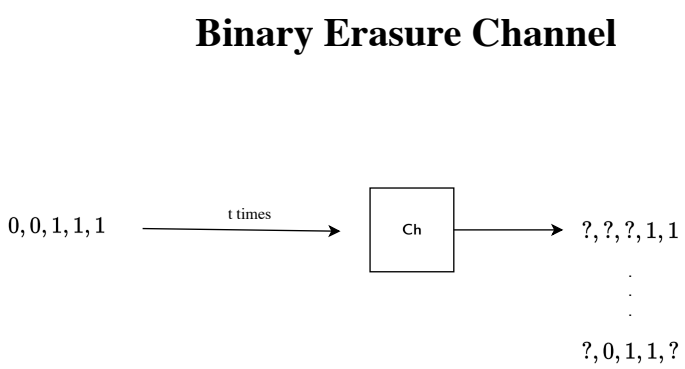


Figure 3.2: Binary Erasure Channel

$\mathcal{Y} = \{0, 1, ?\}$. So, each x_i that passes through Ch transforms into y_i as

$$x_i \rightarrow y_i = \begin{cases} x_i, & \text{with probability } 1 - \varepsilon \\ ?, & \text{with probability } \varepsilon \end{cases}$$

The last channel is the Binary Deletion Channel.

3.2.3 Binary Deletion Channel

The Binary Deletion Channel (BDC) can be seen as a variation of the BEC, but with following difference: while on BEC we know exactly the bits that were erased, on the BDC we do not know the deleted bits. The input and output supports are, respectively $\mathcal{X} = \mathcal{Y} = \{0, 1\}$ To better understand this model, let's look at Fig. 3.3

So, each x_i is transformed in a y_i as

$$x_i \rightarrow y_i = \begin{cases} x_i, & \text{with probability } 1 - \varepsilon \\ \alpha, & \text{with probability, } \varepsilon \end{cases}$$

where α symbolises the empty string. Now that we have described the models, let's define two essential algorithms for trace reconstruction, and one technique to solve them.

3.2.4 Average-Case Reconstruction

Formally, we can define Average Case Reconstruction in the following way.

Binary Deletion Channel

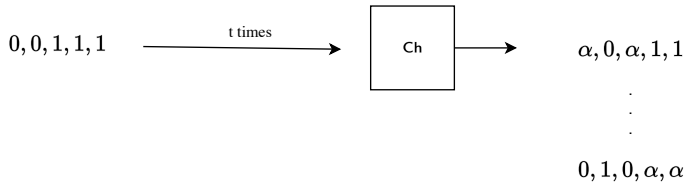


Figure 3.3: Binary Deletion Channel

Definition 3.2.1 (Average Case Reconstruction [16]). *A reconstruction algorithm Rec is said to be (t,ε)- average case for* Ch *if, for a large n, it holds that*

$$2^{-n} \sum_{x \in \{0,1\}^n} \Pr[Rec(Y^{(1)}, Y^{(2)}, \ldots, Y^{(t)}) \neq x] \leq \varepsilon,$$

where $Y^{(1)}, Y^{(2)}, \ldots, Y^{(t)}$ are i.i.d and results from passing the string x t times through the channel Ch.

Batu et al. [2] were the first to obtain results in this direction. They have obtained relevant results from the analysis of the Bitwise Majority Alignment (BMA) algorithm. This algorithm works in a very simple way. Suppose we have t traces $Y^{(1)}, \ldots, Y^{(t)}$ generated by a string $x(x_1, \ldots, x_n)$ that passed through a deletion channel t times. For each trace $Y^{(i)}$, we set a counter $c_i = 1$. Since x has n bits, the BMA will have n rounds, one for each bit x_i of X and, in a given round j we have the following

1. Let m_j be the majority of $Y^{(i)}_{c_i}$ for $i \in [t]$. We set $x'_j = m_j$.

2. For each $i \in [t]$, if $c_i = m_j$, we set $c_i \leftarrow c_i + 1$

3. We set $j \leftarrow j + 1$ and return to Step 1, until $j > n$.

They have proved that the BMA algorithm is an efficient (t, d) average-case reconstruction for the binary deletion channel with $t = O(\log n)$ and $d = O\left(\frac{1}{\log n}\right)$, where t is the number of traces and d the deletion probability of each symbol [2].

In this work, we are interested in analysing different noise models. Average case reconstruction is an interesting scenario to explore to find the minimum traces to reconstruct a string generated uniformly at random, for the lower bound and upper bound cases. In the topic of this work, we will study it to the models described on Section 3.4.

3.2.5 Worst-Case Reconstruction

Definition 3.2.2 (Worst-Case Reconstruction [16]). *A reconstruction algorithm Rec is said to be (t,ε)-worst-case if for any $x \in \{0,1\}^n$ it holds that*

$$\Pr[Rec(Y^{(1)}, Y^{(2)}, \ldots, Y^{(t)}) \neq x] \leq \varepsilon,$$

where each $Y^{(1)}, Y^{(2)}, \ldots, Y^{(t)}$ results from passing the string x ,t times, through the channel Ch.

The first works on Worst Case Reconstruction were also accomplished in [2], with the BMA. The proposed (t, ε)- worst-case reconstruction algorithm for the binary deletion channel, needs $t = (n \log n)$ traces, and a deletion probability $d = \Omega(n^{-(1/2+\varepsilon)})$, where $\varepsilon > 0$ is an arbitrary constant. This result was later improved in [6], where the worst-case reconstruction algorithm, is poly(n) in the number traces and the deletion probability $d = n^{-(1/3+\varepsilon)}$. A worst-case reconstruction algorithm is described as follows. Just like in the Average-Case Reconstruction, we want to analyse, in this work, the lower and upper bounds of Worst-Case Reconstruction over the models described on section 3.4.

3.2.6 Mean-Based Reconstruction

Mean-based reconstruction technique gained recognition after Holenstein et al. in [10] used this reconstruction approach to show exists a (t, ε)-worst case reconstruction algorithm, with $t = \exp(\tilde{O}(\sqrt{n}))$ and $\varepsilon = e^{-\omega(n)}$ in a BDC. This kind of algorithm can be thought of as an algorithm that only requires knowledge of the expected value of each trace coordinate. More formally, it can be defined as this.

Definition 3.2.3 (Mean-based Reconstruction [7]). *Let $Y_x = (Y_{x,1}, Y_{x,2}, \ldots)$ denote the trace distribution on input $x \in \{0,1\}^n$. The mean trace μ_x is given as*

$$\mu_x = (E[Y_{x,1}], E[Y_{x,2}], \ldots).$$

To estimate μ_x we will need to calculate the empirical mean $\hat{\mu}$, for all t traces $T^{(1)}, T^{(2)}, \ldots, T^{(t)}$, sampled i.i.d. according to Y_x

$$\widehat{\mu_i} = \frac{1}{t} \sum_{j=1}^{t} T_i^{(j)}, i = 1, 2 \ldots$$

The goal is to output a string $\hat{x}$, which minimizes $\|\mu_{\hat{x}} - \widehat{\mu}\|_1$. One example of application of this algorithm is in the derivation of the optimal upper bound of the number traces needed for trace reconstruction over a binary symmetric channel, which we will describe in section 3.2.1 and to the models that we will explore in this work, which are described in section 3.4.

These were the definitions of the principal reconstruction algorithms. Now we are in conditions to define the sample complexity, which is the optimal number of traces needed for efficient reconstruction, for the upper bound of the BSC and BEC and analyse some results for the BDC.

3.2.7 Upper bound for Binary Erasure Channel

To analyse this model, we need to consider the model's properties. Suppose we want to reconstruct x with high probability $1-\gamma$. For every coordinate i, we know x_i if it was not erased in all traces. If x_i is erased in all traces, the following probability holds

$$\Pr(x_i \text{ to be erased in all traces}) = \varepsilon^t.$$

If we want this probability to be $\leq \gamma$, we have

$$\varepsilon^t \leq \frac{\gamma}{n}, \tag{3.2}$$

Reorganizing this inequality, we conclude that it suffices to have

$$t \geq \frac{\log(n/\gamma)}{\log(1/\varepsilon)}. \tag{3.3}$$

Since $\gamma = \frac{1}{n}$, this inequality is satisfied if we take

$$t = O(\log(n)). \tag{3.4}$$

If we apply a union bound, we can derive the upper bound of the number of traces needed to recover all bits.

$$\begin{aligned}
&\Pr(x_1 \text{ to be erased in all traces} \vee \cdots \vee x_i \text{ to be erased in all traces}) \\
&\leq \sum_{i=1}^{n} \Pr(x_i \text{ to be erased in all traces}) \\
&= n\varepsilon^t.
\end{aligned}$$

We want this probability to be $\leq \gamma$. So the following inequality holds

$$n\varepsilon^t \leq \gamma. \tag{3.5}$$

Reorganizing the whole expression, we get

$$t \geq \frac{\log(n/\gamma)}{\log(1/\varepsilon)}. \tag{3.6}$$

As before, setting $\gamma = \frac{1}{n}$, we get

$$t = O(\log(n)). \tag{3.7}$$

3.2.8 Upper bound for Binary Symmetric Channel

Here, we will apply the mean-based reconstruction algorithm to derive the upper bound for the BSC, to reconstruct the input string x with high probability $1-\gamma$. Before using it,we must define some important variables that will help us derive an upper bound of

the number of traces. First, we calculate the empirical mean of all y_i, the $\overline{Y_i}$ which is going to be our unbiased estimator of $E[Y_i] = \mu_i$.

$$\overline{Y}_i = \frac{\sum_{j=1}^{t} y_i^{(j)}}{t}.$$

Assuming, without loss of generalization that $x_i = 1$, the $E[Y_i]$ is

$$E[Y_i] = 1 - \varepsilon$$

Since we use a mean-based reconstruction algorithm, we can use a Chernoff Bound to achieve our goal, namely eq. (2.4).

Due to the nature of the channel, each Y_i is independent. We fail reconstructing each x_i with probability $\frac{\gamma}{n}$. So we can write the whole expression as

$$\Pr\left(|\overline{Y}_i - \mu_i| \geq \delta\mu_i\right) \leq \frac{\gamma}{n}.$$

Applying the Chernoff bound to calculate the upper bound, we set

$$\Pr\left(|\overline{Y}_i - \mu_i| \geq \delta\mu_i\right) \leq 2e^{-\delta^2\mu_i/3.}$$

Multiplying $\overline{Y}_i$, μ_i, and $\mu_i\delta$ by t, we have

$$\Pr\left(|\overline{Y}_i t - \mu t| \geq \delta\mu t\right) \leq 2e^{-\delta^2\mu t/3.} \tag{3.8}$$

We need to calculate the δ parameter to give the tightest upper bound. Keeping this in mind, we write

$$\begin{aligned} \delta E[Y_i] &< E[Y_i] - \frac{1}{2} && (3.9)\\ &< \frac{1}{2} - \varepsilon && (3.10)\\ &< \frac{1}{2}. && (3.11) \end{aligned}$$

Given this, and readjusting the inequality, we say that

$$\delta < \frac{1}{2(1-\varepsilon)}. \tag{3.12}$$

Recalling that we fail to reconstruct each x_i with probability $\frac{\gamma}{n}$, solving the following equation will give us the desired upper bound

$$\begin{aligned} 2e^{-\frac{1}{4(1-\varepsilon)^2}t/3} &\leq \frac{\gamma}{n} && (3.13)\\ & && (3.14) \end{aligned}$$

Reorganizing the whole inequality, we can say that

$$t \geq 12(1-\varepsilon)^2 \frac{\log\left(\frac{2n}{\gamma}\right)}{\log(e)} \tag{3.15}$$

Setting $\gamma = \frac{1}{n}$, we get

$$t = O(\log(n)) \tag{3.16}$$

This is the upper bound of needed traces to reconstruct the i^{th} bit with a probability $1 - \frac{\gamma}{n}$. To reconstruct the whole string, we should do the following

$$\Pr(|\overline{Y_1} - \mu| \geq \delta\mu \vee \cdots \vee |\overline{Y_i} - \mu| \geq \delta\mu) \tag{3.17}$$

$$\leq \sum_{i=1}^{n} \Pr(|\overline{Y_i} - \mu| \geq \delta\mu) \tag{3.18}$$

$$\leq 2ne^{-\frac{1}{4(1-\varepsilon)^2}t/3}. \tag{3.19}$$

We fail to reconstruct the whole string with probability γ. So, we can write

$$2ne^{-\frac{1}{4}(1-\varepsilon)t/3} \leq \gamma. \tag{3.20}$$

$$\tag{3.21}$$

Rearranging the expression, we say

$$t \geq 12(1-\varepsilon)^2 \frac{\log\left(\frac{2n}{\gamma}\right)}{\log(e)}. \tag{3.22}$$

Taking $\gamma = \frac{1}{n}$, we get

$$t = O(\log(n)). \tag{3.23}$$

3.2.9 Binary Deletion Channel Results

Some prior work has been done to derive worst-case, average-case and mean-based reconstruction algorithms over deletion channels. Chase [5] has developed an upper bound on worst-case reconstruction, with a (t, ε)-worst-case reconstruction algorithm, where $t = \exp(\tilde{O}(n^{1/5}))$ and $\varepsilon \in (0, 1)$. The lower bound for worst-case reconstruction was demonstrated in [4], for $t = \Omega(n^{3/2}\log^7(n))$ and $\varepsilon \in (0, 1)$.

Concerning average-case reconstruction, the latest upper bound was demonstrated in [9], where $t = \exp(O(\log^{1/3}(n)))$ and $\varepsilon \in [0, 1)$ and the lower bound in [4], for $t = \Omega\left(\frac{\log^{5/2} n}{(\log\log n)^7}\right)$ and $\varepsilon \in (0, 1)$. Finally, in terms of mean-based reconstruction, it has been applied to show that, for a generalization type of deletion channel called oblivious synchronization channel, the achieved upper bound was $\exp(O(n^{1/3}))$ [7]. The resulting upper bounds are exponential, while the lower bounds are linear values.

As already shown before, each x_i is deleted with probability ε in this channel type. Contrary to the binary erasure channel, the receiver will not know the position of the deleted bits, making it difficult to do the same procedure as in the BEC. The mean-based reconstruction approach that we use in the BSC, will not work here as well. How each coordinate is deleted with a given probability, many x_i will not be in the correct order in the

generated traces. This will lead to when we apply the mean-based reconstruction algorithm, the empirical means over each coordinate will not be correct, giving us incorrect values. In this dissertation, we will study several biological models. Some of their properties have similarities to the channels described above and some of these reconstruction algorithms can be applied to reconstruct, from corrupted samples, the original string.

Now that we have instantiated the three main channels, we can compare all of them.

3.3 Comparative Analysis of Models and Algorithms

As already have been discussed in the previous section, different approaches and strategies are needed for different models and settings. In the BSC described in Section 3.2.1, we could apply directly the mean-based reconstruction algorithm, because we know all the bit positions, calculating the expected value coordinate by coordinate.

Relatively to the BEC, which is in Section 3.2.2, we use the model properties to derive the upper bounds, because the same bit is maintained on each coordinate, but is changed for a "?", with a given error probability. If we analyse each coordinate over all traces, we do not know if the "?" corresponds to a "0" or "1", so techniques like mean-based reconstruction will not work here. So, different methods were developed to solve this problem, namely, linear programming, to find the sample complexity, as demonstrated in [18].

The most challenging model is the binary deletion channel. How each bit is deleted with a given probability, direct application of mean-base reconstruction or other simpler approaches, will not work, because, if we do not know the exact position of the deleted bits, we have synchronisation errors. When calculating the empirical mean in this case, we will get the wrong mean estimates. So, we must resort to complex analysis, as described in [7, 19].

3.4 Proposed Work

Trace reconstruction techniques have received a lot of attention over the years, due to their theoretical results. Levenshtein [13] [14], have proved some theoretical results over a substitution channel, where the symbols in strings are mutated independently, like in the binary symmetric channel. In the following years, Batu et al. [2], the BMA algorithm was developed and applied to the binary deletion channel. So, computational biologists have tried to create a practical approach of these models to the DNA, in the context of immunogenomics and DNA data storage [3]. The motivation here is to take a simplified biological problem, to develop algorithms, which can lead us to solutions to very complex problems. The inspiration for our thesis work is the open problems in immunogenomics presented in [3], based on the recombination of VDJ genes, which are presented in all vertebrate immunologic systems. We will apply trace reconstruction to algorithms to try to find the sample complexity for upper and lower bounds, in the order of, respectively

$O(n \log(n))$ and $\Omega(n \log(n)$. The DNA sequences have alphabet $\mathcal{A} = \{A, C, G, T\}$, but, in our study, we will only work over the binary alphabet i.e., over the alphabet $\mathcal{X} = \{0, 1\}^n$ to study the models. A DNA sequence can be interpreted as a string s of length $|s|$. The operations to generate traces from the original string are the following.

- $Trim(s)$: A pair of integers k and l are sampled uniformly at random, fulfilling the condition $k + l \leq |s|$. The prefix of the string of length l and the suffix of length k are trimmed. This is similar to what a Binary Deletion Channel does. This is represented in Fig. 3.4.

- $Mutate_\varepsilon(s)$: Each bit in s is flipped with a given probability ε, to the opposite bit. This is exactly what a Binary Symmetric Channel does, which is exemplified in Fig. 3.5.

- $Extend_t(s)$: Two strings R_1 and R_2, whose length is generated uniformly at random from a interval $[0, t]$, where t is fixed integer. We concatenate them with s like $R_1 \mid\mid s \mid\mid R_2$, where $\mid\mid$ is the concatenation operator. An exemplification of this process can be seen in Fig. 3.6.

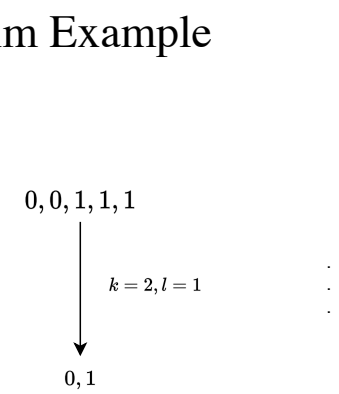


Figure 3.4: Example of trim application.

In the case of models that we only want to trim or extend the suffix of s, the operations are TrimSuffix and ExtendSuffix.

From these operations, several types of models can be derived. In this work, we will study trace reconstruction and population recovery algorithms in five models. Lets define each of them separately.

3.4.1 TrimSuffixAndExtend

Mutate Example

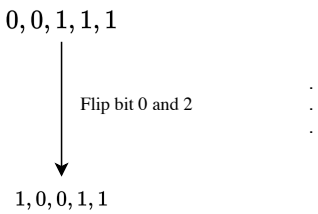


Figure 3.5: Example of mutate application.

Extend Example

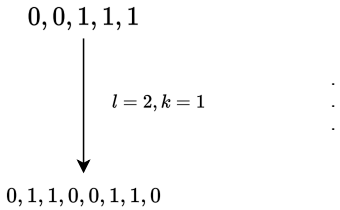


Figure 3.6: Example of an extend operation.

TrimSuffixAndExtend

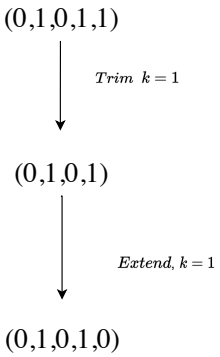


Figure 3.7: TrimSuffixAndExtend model.

Fig. 3.7 shows a concrete example of TrimSuffixAndExtendModel. This type of model results from sampling uniformly at random an integer $k \in [0, |s|]$. Then the suffix of length k is trimmed in s and a string of length k is concatenated.

3.4.2 TrimAndExtend

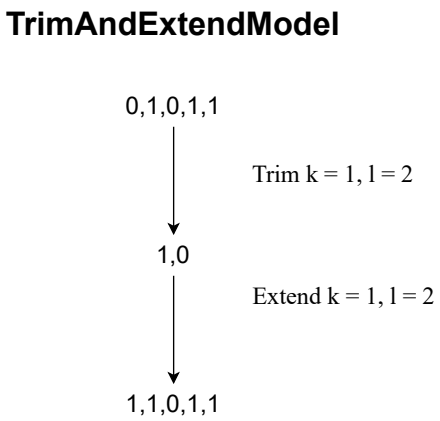


Figure 3.8: TrimAndExtend model

The Fig. 3.8 illustrates how the TrimAndExtend model works. In this scenario, a pair of non-negative integers (k, l) are sampled uniformly at random, such that $k + l \leq |s|$. The prefix of length k and the suffix of length l are trimmed, and two strings r^k and r^l with lengths k and l respectively, are concatenated with the trimmed string, resulting in $r^l||Trim(s)||r^k$.

3.4.3 TrimSuffixMutateAndSuffixExtend

Fig. 3.9 illustrates the TrimSuffixMutateAndSuffixExtedModel. In this model, we first trim the suffix of the string with a given value. Then, with a given probability ε, we mutate each bit of the string. Finally, with a given generated length, we concatenate a string to $Mutate(TrimSuffix(s))||r^l$.

3.4.4 TrimExtendAndMutate

This model is practically similar to Fig. 3.8 except that, after the trim and the extend, each bit is mutated with a given probability. This can be visualized in Fig. 3.10.

Each one of these models exits a corresponding variation. When we trim the suffix or the prefix with length k, the generated suffix or prefix can have a length different from k.

As can be seen, all of these models are combinations of the trace reconstruction models described in sections 3.2.1 to 3.2.3. Keeping this in mind, we are interested in studying the described models, either in the context of trace reconstruction. We want to explore

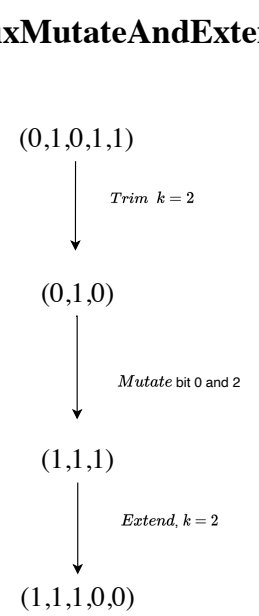


Figure 3.9: TrimSuffixMutateAndSuffixExtendModel.

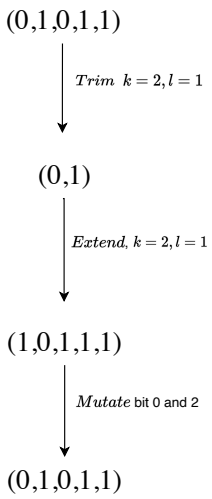


Figure 3.10: TrimExtendAndMutateModel.

the different algorithm approaches to do worst-case and average-case reconstruction, following the used techniques in the presented literature. With these tools, our goal is to find the minimum traces needed for upper and lower bounds on models and conceptualise efficient algorithms to solve the problems.

3.5 Genetic Algorithm

A genetic algorithm is an algorithm inspired by Charles Darwin's survival theory, which was proposed by J.H Holland in 1992 [11]. This theory suggests that only the fittest individuals will survive, reproduce and pass their information to the offspring, resulting in the next generation being better suited to survive in a given environment.

A genetic algorithm is made up of chromosomes, a fitness function, and biological operators. These components work together to determine the optimal solution. Chromosomes represent the possible solutions, the fitness function determines how well each chromosome works, and biological operators mimic natural selection and mutation to create new, improved chromosomes.

Overall, the genetic algorithm works by creating a population of chromosomes, evaluating their fitness, selecting the fittest individuals, and using them to create the next generation. This process is repeated until the optimal solution is found.

3.5.1 Chromossomes

Inspired by biological chromosomes, these artificial chromosomes, also known as individuals, are represented as binary strings. Each position on a chromosome, also known as a locus, has two possible alleles, 0 or 1 [11]. These chromosomes are the key elements in the solution space and are manipulated by genetic operators to generate new chromosomes. We use a fitness function to evaluate the chromosomes and determine which ones are most suitable to reproduce in the next generation.

3.5.2 Fitness Function

The fitness function is an optimization function which is used to assign a value to each gene in the population. According to the problem, the values for the fitness function are higher on the most appropriate chromosomes and lower on the lesser ones. After assigning a fit value to each chromosome, we should select the best ones. This can be done using selection methods.

3.5.3 Selection

In this phase, chromosomes are selected for reproduction according to their fitness function. There are many selection methods, like Roulette wheel selection, Rank Selection, and Tournament Selection. But the one we think is the most appropriate for our problem and

we will describe in detail is the tournament selection [22]. Algorithm 2 represents the pseudo-code for this selection method.

Algorithm 1 Tournament algorithm [22]

```
1: function TOURNAMENT(T , k)
2:     T ← List of individuals randomly selected from a population
3:     k ← Number of elements in T
4:     best ← T[0]
5:     for i = 1 to k do
6:         individual ← T[i]
7:         if Fitness(best) < Fitness(individual) then
8:             best ← T[i]
       return best
```

Algorithm 2 Tournament selection algorithm [22]

```
1: function TOURNAMENTSELECTION(P,k,n)
2:     P = Number of individuals in the population
3:     k = Tournament Size
4:     n = Number of individuals to be selected
5:     T ← []
6:     Bests ← []
7:     for i = 0 to k  do
8:         Select n elements at random from P and add them to T
9:         B[i] ← Tournament(P, k)
10:        T ← []
       return Bests
```

After selecting the best individuals, which we will denominate as parents, let's do some biological operations called crossover operators, to produce offspring.

3.5.4 Crossover Operators

Crossover operators are the main tools to generate offspring, by combining information from parents. The most known crossover operators are the single-point, two-point, uniform, shuffle and scattering. For the sake of efficiency, and not compromising the algorithm's performance, we chose the scattering crossover operator. The following pseudocode shows how this operator works

After generating all the offspring, mutation operators are introduced to maintain the diversity in the population [11].

3.5.5 Mutation Operator

This operator is responsible for maintaining the diversity between all the individuals. One of the most common mutation operators is the simple inversion. This operator merely randomly inverts the bits in the string. Besides its importance, introducing this operator

Algorithm 3 Crossover operator

```
function Scattered Crossover(p1,p2)
    offspring1 ← []
    offspring2 ← []
    vec ← [random() < 0.5 for i in range(len(p1))]
    for i in range(length(p1)) do
        if vec[i] > 0.5 then
            offspring1[i] ← p1[i]
            offspring2[i] ← p2[i]
        else
            offspring2[i] ← p1[i]
            offspring1[i] ← p2[i]
    return offspring1,offspring2
```

in the context of trace reconstruction will increase the error in the traces, getting a worse estimation of the input.

These operators will be executed over a pre-defined number of generations, returning to the final one, the global optimal solution. The following pseudo-code illustrates how a classical genetic algorithm works.

Algorithm 4 Classical Genetic Algorithm [11]

```
1:  function GeneticAlgorithm(n,maxIt)
2:      n = Population Size
3:      maxIt = maximum number of generations
4:      t ← 0
5:      pop ← generate a population of size n
6:      fits ← fitness values of each chromosome
7:      best ← best fitness value in fits
8:      while t < maxIt do
9:          Select a pair of chromosomes from pop for reproduction
10:         Apply a crossover operator
11:         Apply a mutation operator
12:         Replace the old population by the new one
13:         fits ← []
14:         Computes the fitness values for the new population and adds them to fits
15:         t ← t + 1
16:         update best
        return best
```

4

TrimSuffixAndExtend

In this chapter, we derive the upper and lower bounds for the TrimSuffixAndExtendModel, described in section 3.4.1. To determine the upper bounds, we utilized an algorithm that identifies the most frequent trace in a set of traces. When establishing the lower bounds, our initial focus will be on recovering x_n since it is the bit that presents the highest level of difficulty to recover.

4.1 Upper Bounds

In this section, we will derive the upper bounds for the TrimSuffixAndExtendModel. Firstly, we tried to apply mean-based to the last bit, since it is the hardest bit to recover. Then, we apply an algorithm that finds the trace that appears more often in a trace set, in a way to guarantee that the trace is the original string x.

Let's first discuss how this model works. Suppose we have an arbitrary bit string $x = (x_1, x_2, \ldots, x_n)$ that passes t times through a channel, generating t traces $(y^{(1)}, y^{(2)}, \ldots, y^{(n)})$. Each trace is generated by sampling a trim suffix k uniformly at random, trimming x, generating a new suffix of length k and concatenating it to the resulting string, forming the trace. Let Y_x be the random variable which represents a trace and K the random variable which represents a trimming suffix.

Before introducing this algorithm, let's apply mean based to the last bit to see which lower bound we can get. Each x_i is transformed in a y_i in the following way

$$x_n \longrightarrow y_n = \begin{cases} x_n, \text{with probability } (1 - \frac{1}{n+1})0.5 + \frac{1}{n+1} \\ 1 - x_n, \text{with probability } (1 - \frac{1}{n+1})0.5 \end{cases}$$

Assuming, without loss of generality, that $x_n = 1$, the expect value of Y_{x_n} is

$$E[Y_{x_n}] = (1 - \frac{1}{n+1})0.5 + \frac{1}{n+1} = \frac{n+2}{2(n+1)}. \tag{4.1}$$

We will use Equation (2.4) to estimate the lower bound. First, we will calculate the empirical mean, which is our unbiased estimator.

$$\overline{Y}_{x_n} = \sum_{i=1}^{n} \frac{y_n^{(t)}}{t} \tag{4.2}$$

Now, we are in conditions to apply the Chernoff Bound.

$$\Pr(|\overline{Y}_{x_n} - E[Y_{x_n}]| \geq \delta E[Y_{x_n}]) \leq 2e^{-\delta^2 E[Y_{x_n}]t/3}. \tag{4.3}$$

We fail to reconstruct each bit with probability, at most, $\frac{\gamma}{n}$. Given this fact, we write

$$2e^{-\delta^2 E[Yx_n]/3} \leq \frac{\gamma}{n}. \tag{4.4}$$

Now, we must find the δ parameter. To do this, we say that

$$\delta E[Y_{x_n}] < E[Y_{x_n}] - \frac{1}{2} \tag{4.5}$$

$$< \frac{1}{2(n+1)}, \tag{4.6}$$

so we can say that δ is

$$\delta < \frac{1}{n+2}. \tag{4.7}$$

Substituting this value in eq. (4.4), we have

$$2e^{-\frac{1}{(n+2)^2}\frac{n+2}{2(n+1)}t/3} \leq \frac{\gamma}{n} \tag{4.8}$$

$$\tag{4.9}$$

Rewriting the whole inequality, we conclude that

$$t \geq 6(n+1)(n+2)\frac{\log(2n/\gamma)}{\log(e)}. \tag{4.10}$$

Considering $\gamma = \frac{1}{n}$, we can state that

$$t = O(n^2 \log(n)). \tag{4.11}$$

We can conclude that to recover the last bit, the number of traces is upper bound by $O(n^2 \log(n))$. We believe we can achieve a better upper bound, so we tried another approach to get an upper bound of $O(n \log(n))$ traces. This is when the most common trace algorithm comes in. The following theorem states this is the best we can do.

Theorem 4.1.1. *There exists a (T, ε)-worst-case trace reconstruction algorithm for the TrimSuffixAndExtend model with $T = 2n \log n$ and $\varepsilon = o(1/n)$.*

To prove our theorem, let's see the following lemmas, which will be essential to understanding the proof of the theorem.

Lemma 4.1.1. *Given an arbitrary string x,* $\Pr(Y_x = x) = \frac{2-2^{-n}}{n+1}$.

Proof. Applying Theorem 2.1.1 we have

$$\Pr(Y_x = x) = \sum_{i=0}^{n} \Pr(Y = x|K = i) \cdot \Pr(K = i) \tag{4.12}$$

$$= \frac{1}{n+1} \sum_{i=0}^{n} \Pr(Y_x = x|K = i) \tag{4.13}$$

$$= \frac{1}{n+1} \sum_{i=0}^{n} \frac{1}{2^i} \tag{4.14}$$

$$= \frac{2-2^{-n}}{n+1}. \qquad \square$$

Now, we can formulate the lemma for $\Pr(Y_x = y), y \neq x$.

Lemma 4.1.2. *Given arbitrary strings x and $y \neq x$, it holds that* $\Pr[Y_x = y] \leq \frac{1-2^{-n}}{n+1}$

Proof. Let r be defined as

$$r = \min\{i : x_i \neq y_i\}.$$

Fig. 4.1 illustrates how we can calculate r.

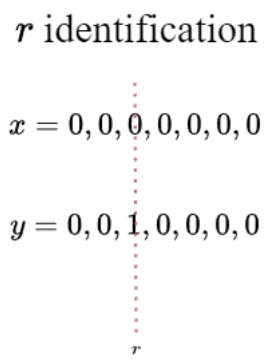


Figure 4.1: Example of r identification.

Given this, we must apply conditions to k, as a function of n and r.
If $r < n - k$ then

$$\Pr(Y_x = y|K = k) = 0, \tag{4.15}$$

and if $r \geq n - k$ then

$$\Pr(Y_x = y|K = k) = 2^{-k}. \tag{4.16}$$

Taking this, we have

$$\Pr(Y_x = y) = \sum_{i=n-r}^{n} \Pr(Y = y|K = i) \cdot \Pr(K = i) \tag{4.17}$$

$$= \frac{1}{n+1} \sum_{i=n-r}^{n} \frac{1}{2^i} \tag{4.18}$$

$$= \frac{2^{r-n+1}(1-2^{r+1})}{n+1} \tag{4.19}$$

$$\leq \frac{1-2^{-n}}{n+1}. \qquad \square$$

Let's assume that we have a total of N traces, where N_x are the number of traces that are equal to the input string x and N_y the number of traces which correspond to the string $y, y \neq x$. Since we want to recover the string x, we wish to show that

$$\Pr(N_x > N_y, \text{ for all } y \neq x) \geq 1 - \frac{1}{n}.$$

To solve this, we will apply a Chernoff bound, namely eq. (2.2) and eq. (2.3), but first, we must calculate $E[N_x]$ and $E[N_y]$.

$$E[N_x] = N \cdot \frac{(2-2^{-n})}{n+1}$$

and

$$E[N_y] = N \cdot \frac{(1-2^{-n})}{n+1}.$$

Now, we will set $N = 2(n+1)\log n$. Using this, and assuming a big n, we can upper bound both expressions. Let's first upper bound $E[N_x]$.

$$E[N_x] = 2(n+1)\log(n)\frac{(2-2^{-n})}{n+1} \tag{4.20}$$

$$> 3\log(n). \tag{4.21}$$

Now, we will upper bound $E[N_y]$.

$$.E[N_y] = 2(n+1)\log(n)\frac{(1-2^{-n})}{n+1}$$

$$< 2\log(n).$$

For our algorithm to work, we must have $N_x > N_y$, for all $y \neq x$. For this, we will estimate the probability of N_x and N_y being bigger than $\frac{5}{2}\log(n)$. To study this, we must apply an upper bound for $\Pr(N_x > \frac{5}{2}\log n)$ and a lower bound for $\Pr(N_y > \frac{5}{2}\log n)$. N_x and N_y can be viewed as a sum of 0-1 random variables, so, we are in conditions to apply the Chernoff bound. Applying eq. (2.3) on the first one, we have

$$\Pr(N_x > \frac{5}{2}\log n) < e^{-(E[N_x]/36)/3}. \tag{4.22}$$

Because we want to lower bound this probability, we rewrite

$$\Pr(N_x \leq \frac{5}{2}\log n) \geq 1 - n^{-\frac{1}{36\ln(2)}}. \tag{4.23}$$

Applying the same to the second one, we get

$$\Pr(N_y > \frac{5}{2}\log n) < e^{-(E[N_y]/16)/3} \tag{4.24}$$

that can be simplified to

$$\Pr(N_y > \frac{5}{2}\log n) < n^{\frac{-1}{24\ln(2)}} \tag{4.25}$$

To prove that our algorithm works, we must apply a union bound. The algorithm will fail if $N_x \leq 3\log n$ or there is a trace $y \neq x$ that appears among the N traces more than $3\log n$ times. More precisely, we have the upper bound

$$\begin{aligned}
\Pr(\text{Algorithm fails}) &\leq \Pr(N_x \leq \frac{5}{2}\log n \text{ or there is a trace } y \neq x \text{ such that } N_y > \frac{5}{2}\log(n)) \\
&\leq \Pr[N_x \leq \frac{5}{2}\log n] + \Pr(\text{there is a trace } y \neq x \text{ such that } N_y > \frac{5}{2}\log(n)) \\
&\leq \Pr(N_x \leq \frac{5}{2}\log n) + \sum_{i=1}^{N} \Pr(\text{i-th trace is } \neq x \text{ such that } N_y > \frac{5}{2}\log(n)) \\
&\leq n^{-\frac{1}{36\ln(2)}} + 2(n+1)\log(n) \cdot n^{\frac{-1}{24\ln(2)}}.
\end{aligned} \tag{4.26}$$

From Equation (4.26), we can conclude that our algorithm fails with probability $o(1/n)$, proving that our algorithm works, with high probability.

4.2 Lower Bounds

This section will demonstrate the Lower Bounds needed for the TrimmSuffixAndExtend-Model.

Theorem 4.2.1. *Every (T, ε)-worst-case trace reconstruction algorithm for the TrimSuffixAndExtend model will need, at least, $T = \Omega(n\log(n))$ traces to reconstruct the input string with error probability $\varepsilon \in (0, \frac{1}{4})$.*

To derive this, we must focus on x_n since it is the most demanding element of x to recover.

Lemma 4.2.1. *The number of T traces needed is, at least,$\Omega(n\log(n))$.*

Proof. Let E be the event "x_n is always trimmed". When E occurs, any algorithm fails with probability $\frac{1}{2}$. As we already have seen in Section 4.1, x_n is trimmed with probability

$\left(1 - \frac{1}{n+1}\right)$. If we have T traces, we can say that the probability of x_n being trimmed in all of them is $\left(1 - \frac{1}{n+1}\right)^T$. If the algorithm fails with probability δ, then

$$\frac{P(E)}{2} \leq \delta \tag{4.27}$$

$$P(E) \leq 2\delta. \tag{4.28}$$

Substituting $P(E)$, we have

$$\left(1 - \frac{1}{n+1}\right)^T \leq 2\delta, \tag{4.29}$$

which we can rewrite as

$$T \ln\left(1 - \frac{1}{n+1}\right) \leq \ln(2\delta) \tag{4.30}$$

$$T \geq \frac{\ln(1/2\delta)}{\ln(1+1/n)}. \tag{4.31}$$

Taking into account the inequality $1 + x \leq e^x$, we can upper bound $\ln(1 + 1/n)$ as

$$\ln(1+1/n) \leq \ln(e^{1/n}) \tag{4.32}$$

$$\ln(1+1/n) \leq 1/n. \tag{4.33}$$

Substituting in Equation (4.30), we have

$$T \geq \frac{\ln(1/2\delta)}{1/n} \tag{4.34}$$

$$\geq n \ln(1/2\delta). \tag{4.35}$$

□

5

TrimSuffixAndExtendWithMutations

In this chapter, we started an attempt to analyse the upper bounds for the TrimSuffixAndExtendWithMutations model, which is explained in Section 3.4.3. To obtain the upper bounds, we have developed an algorithm that analyzes all traces in stages, calculates the average coordinate by coordinate, and then determines the Hamming distance between the mean values and the ones on the traces. If the calculated distance is below a given threshold, the trace will be selected for the next stage of the algorithm. Otherwise, it will be rejected. This process is repeated several times until we predict the entire input string. The lower bound follows from the lower bound from the TrimSuffixAndExtend model.

5.1 Initial Step For Upper Bound Calculation

This section presents the beginning of the attempt to try to analyse the upper bounds for the TrimSuffixAndExtendWithMutations model.

As we did in section 3.4.1, we will start by applying mean-based reconstruction to x_n and see which upper bounds we can get. After this, we will start the formal analysis of our developed algorithm. This algorithm will, iteratively, process a window length of all traces, predict a specific part of the input string, compute the Hamming distance, and discard traces above a threshold. The algorithm stops when it predicts the entire input string.

Let's first describe how this model works. Suppose we have a bit string $x = (x_1, x_2, \ldots, x_n)$ that goes through a channel that produces t traces, denoted as $(y^{(1)}, y^{(2)}, \ldots, y^{(t)})$. Let's denote the trimming suffix as k, the mutation rate as γ and Y a random variable which represents a trace of x.

Now that we have described our model, we will apply mean-based reconstruction to the last bit x_n. Each x_n is transformed in a y_n as

$$x_n \to y_n = \begin{cases} x_n, \text{with probability } (1 - \frac{1}{n+1})\ 0.5 + \frac{1}{n+1}(1-\gamma) \\ 1 - x_n, \text{with probability } (\frac{1}{n+1})\gamma + (1 - \frac{1}{n+1})\ 0.5 \end{cases}$$

We assume, without loss of generality, that $x_n = 1$. Given this assumption, we get

$$E[Y_{x_n}] = (1 - \frac{1}{n+1})0.5 + \frac{1}{n+1}(1-\gamma) \tag{5.1}$$

$$= \frac{n+2}{2(n+2)}. \tag{5.2}$$

Once again, we will calculate the empirical mean and then use it on the Chernoff Bound.

$$\overline{Y}_{x_n} = \sum_{i=1}^{n} \frac{y_n^{(t)}}{t}. \tag{5.3}$$

Given this, we write

$$\Pr(|\overline{Y}_{x_n} - E[Y_{x_n}]| \geq \delta E[Y_{x_n}]) \leq 2e^{-\delta^2 E[Y_{x_n}]t/3}. \tag{5.4}$$

Now we must calculate δ

$$\delta E[Y_{x_n}] < E[Y_{x_n}] - \frac{1}{2} \tag{5.5}$$

$$< \frac{1}{2(n+1)}. \tag{5.6}$$

With this, we can say that δ is

$$\delta < \frac{1}{n+2}. \tag{5.7}$$

Substituting this on the Chernoff Bound, we get

$$2e^{\frac{1}{(n+2)^2}\frac{n+2}{2(n+1)}t/3}. \tag{5.8}$$

We fail to reconstruct x_n with probability at most $\frac{\alpha}{n}$. Given this, we write

$$2e^{\frac{1}{(n+2)^2}\frac{n+2}{2(n+1)}t/3} \leq \frac{\alpha}{n}. \tag{5.9}$$

Reorganizing this inequality, we will state that

$$t \geq 6(n+2)(n+1)\frac{\log(2n/\alpha)}{\log(e)}. \tag{5.10}$$

Since $\alpha = \frac{1}{n}$, we get

$$t = O(n^2 \log(n)). \tag{5.11}$$

Applying mean-based reconstruction to the last bit will lead to the upper bound $O(n^2 \log(n))$. We think we can do better than this, so we designed a new algorithm to try to get an upper bound of $O(n \log^2(n))$ traces.

Conjecture 5.1.1. *There exists a (T, ε) - worst-case reconstruction algorithm for the TrimSuffixAndExtend with mutations model with $T = 20n\log^2(n)$ and $\varepsilon = o(1/n)$*

We need to define some terms and prove some lemmas that demonstrate the limits of our approach. So, let's dive into the details of how this works.

Before entering the algorithm itself, we must specify a crucial definition, which will help us, in each iteration, to define which traces are "good", based on the Hamming distance, according to definition 3.1.5, between the already recovered length and the same trace length.

Definition 5.1.1. *Suppose we recovered x until position i. We say that a trace is "good" if*

$$\Delta(x'_{i-300\log(n)}, \ldots, x'_i, y_{i-300\log(n)}, \ldots, y_i) < \frac{1}{4} + \frac{\gamma}{2}. \tag{5.12}$$

To better understand the algorithm the following pseudocode shows exactly how it works. First, we need to define the auxiliary functions which will help us build the algorithm.

```
1: function LENGTH(s) return s.size()
```

```
1: function GENERATETRACES(x, γ, nT)
2:     ks ← nT generated trimming suffixes uniformly at random.
3:     Traces ← [][]
4:     for all k in ks do
5:         trimming x in position k
6:         flips each remaining bit independently with prob γ
7:         Generate a new suffix uniformly at random
8:         Adds the trace to Traces
       return Traces
```

```
1: function MEANBASED(i, j, T)
2:     Applies the mean, for all rows of the T, between columns i and j return mean
```

```
1: function HAMMINGDISTANCE(x',t) return absolute difference, coordinate by coordi-
   nate, between x' and t
```

It's important to note that as the trimming position k decreases, the expected Hamming distance between a trace and our prediction increases.We define X as the random variable representing the number of different coordinates between x' and the corresponding coordinates in a trace y.

Now, we analyse between $\frac{n}{2}$ and $\frac{n}{2} - 300\log(n)$. The following lemma bounds the probability of a trace being "good", conditioned by $k < \frac{n}{2} - 300\log(n)$.

Lemma 5.1.1. *Given a trace and trimming position k,*

$$Pr(\textit{trace being good} | k < \frac{n}{2} - 300\log(n)) < n^{-\frac{50}{27\ln 2}}$$

Algorithm 5 TrimSuffixAndExtendWithMutations

```
x ← input string
γ ← mutation rate
n ← length(x)
nT ← 20n log²(n)
T ← generateTraces(x, γ, nT)
i ← 0
j ← n/2
x' ← []
while length(x) != length(x') do
    if j >= 20 log(n) then
        i ← 10 log(n)
        j ← n
    pred ← meanBased(T, i, j)
    x' ← concat(x', pred)
    for all t in T do
        if HammingDistance(x'_{i:j}, t_{i:j}) >= 1/4 + γ/2 then
            T.Remove(t)
    i ← j
    j ← (n+i)/2
```

Proof. To estimate this probability we want to apply a Chernoff bound which estimates how much X surpasses the threshold in definition 5.1.1. When $k < \frac{n}{2} - 300\log(n)$, all bits between $\frac{n}{2}$ and $\frac{n}{2} - 300\log(n)$ are trimmed, so we can say that

$$E[X] = 150\log(n). \tag{5.13}$$

For a trace to be considered "good", it must hold

$$\Delta(x'_{n/2-300\log(n)}, \ldots, x'_{n/2}, y_{n/2-300\log(n)}, \ldots, y_{n/2}) < \frac{1}{4} + \frac{\gamma}{2}. \tag{5.14}$$

We are now in conditions to apply Chernoff Bound. Using eq. (2.2), we have

$$\Pr\left(X > \frac{1}{4} + \frac{\gamma}{2}\right) < e^{-\delta^2 E[X]/3}. \tag{5.15}$$

Now, to find δ, we write

$$(1+\delta)150\log(n) = \frac{1}{4} + \frac{\gamma}{2}$$

Rearranging this equation we can write

$$\delta = \frac{1+2\gamma}{600\log(n)} - 1.$$

Upper bounding δ, the following inequality holds

$$\begin{aligned}\delta &< \frac{1 - 300\log(n)}{300\log(n)} \\ &< \frac{1}{3}.\end{aligned}$$

Rewriting eq. (5.15), we can find the upper bound

$$\Pr(X > \frac{1}{4} + \frac{\gamma}{2}) < e^{-\frac{150\log(n)}{27}} \tag{5.16}$$

$$= n^{-\frac{50}{9\ln 2}}. \tag{5.17}$$

□

From this result, we can derive another lemma.

Lemma 5.1.2. *Given a trace Y which is "good", we can say that*

$$\Pr(k < \frac{n}{2} - 300\log(n) | Y good) \leq 3n^{\frac{-50}{9\ln(2)}}$$

Proof. Applying Theorem 2.1.2, we have

$$\Pr(k < \frac{n}{2} - 300\log(n) | Y\text{good}) = \frac{\Pr(Y\text{good})\Pr(k < \frac{n}{2} - 300\log(n))}{\Pr(Y\text{good})}.$$

If the trim occurs in the last $\frac{n}{2}$ bits, Y will be good with high probability. Given this fact, we can say that $\Pr(Y\text{good}) \geq \frac{2}{5}$.

Considering this fact, we have

$$\begin{aligned}\Pr(k < \frac{n}{2} - 300\log(n) | Y\text{good}) &\leq \frac{5\Pr(Y\text{is good})\Pr(k < \frac{n}{2} - 300\log(n))}{2} \\ &\leq 3n^{-\frac{50}{9\ln(2)}}.\end{aligned}$$

□

6

Experimental Results

This chapter is designated to present the empirical implementation of the algorithms in chapters 4 and 5 and a genetic algorithm in which we will demonstrate we can solve the TrimSuffixAndExtend problem with fewer traces. Besides these two models that we have studied formally, we wanted to test the algorithms we have developed in other proposed models(trimAndExtend, TrimExtendMutate, TrimSuffixExtend with variable suffix length). With them, besides proving that our algorithms work, we want to estimate the average error and show that it is always below the worst-case error. All the experiments were done in Python. We will also show the obtained results and discuss them in the context of our problem.

6.1 Results of Studied Models

In this section, we will present the obtained results for the developed experiments, for the different models.

6.1.1 Most Common Trace Algorithm Results

Here, we applied the most common trace algorithm to estimate the average-case error(see appendix A.1 for the implementation of the algorithm). To estimate this error, we ran the algorithm for input strings from length 64 to 960, with a pace of 128.

We run each input size 300 times and, for each one of them, we sample an input string uniformly at random, generate the traces according to the TrimmSuffixAndExtend model and apply the algorithm to return the prediction. After returning the prediction, we calculate the error as the number of different bits between the prediction and the input string, divided by the input length.

We accumulate this error 300 times and, before passing to the next input length, we calculate the average for the accumulated error, estimating the average-case error for a given input length n. Each blue dot in the figure corresponds to the average-case error for a given input length n.

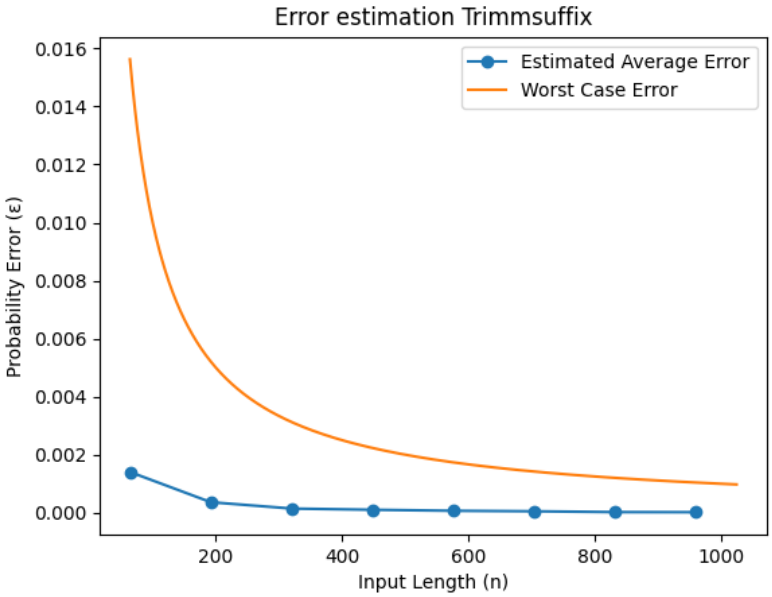


Figure 6.1: Worst case error vs Average case error TrimSuffixAndExtend model

In Fig. 6.1, we used exactly $n \log(n)$ traces to estimate the average-case reconstruction error and show the error as a function of the input length, for the TrimSuffixAndExtend model. As we can see, as n increases, the average error is always below $1/n$, proving that the optimal number of traces is linear in $n \log(n)$. .

6.1.2 Stages Algorithm Results

Here, we can find the results for the partitioned mean-based algorithm (see appendix A.2 for the implementation of the algorithm). The process is similar to the one described in section 6.1.1: we run 300 times the algorithm for each input length from 64 to 1024. The traces are generated according to the trimmSuffixAndExtend with mutations model and the algorithm we use is the one described in chapter 5. We set the mutation rate equal to $\frac{1}{3}$. First, we tried with $n \log^2(n)$ to see if could reconstruct the input string x with high probability.

As we can see in Fig. 6.2, the average-case error is highly above $\frac{1}{n}$, indicating that $n \log(n)$ traces are not sufficient to correctly estimate the input string. Next, we ran the algorithm with $T = 20n \log^2(n)$ traces for the same mutation rate.

As we can see in Fig. 6.3, as n increases, the average case error also decreases and it is always below $\frac{1}{n}$. We prove that $20n \log^2(n)$ traces are sufficient to correctly estimate the input string x.

6.1.3 Genetic Algorithm Results

The genetic algorithm is introduced here to see if we can improve the optimal number of traces and to make some experiments on the models we have not rigorously analysed

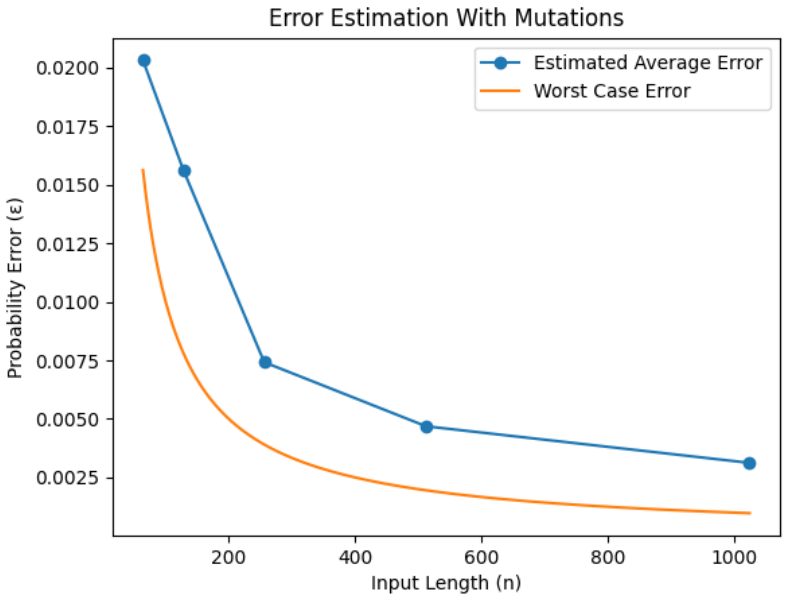


Figure 6.2: Worst-case vs average-case error $T = n \log^2(n)$

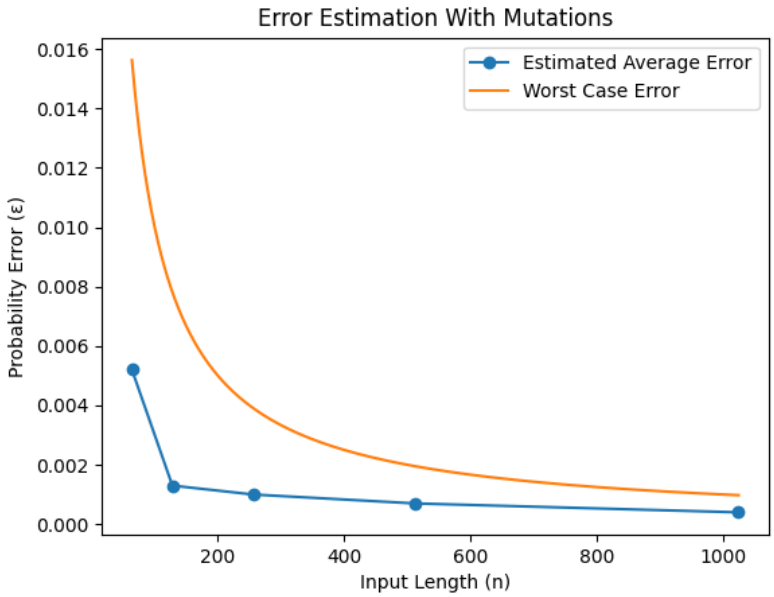


Figure 6.3: Worst case error vs Average case error $T = 20n \log^2(n)$

(TrimAndExtend, TrimandExtend with mutations and TrimSuffixExtend with variable suffix). For the whole implementation of this algorithm see appendix A.3.

Let's first try to improve the number of traces for the TrimSuffixAndExtend with the mutations model. Once again, we ran 300 times the algorithm for input lengths from 64 to 1024. The parameters we chose for the genetic algorithm, were 20 generations, $T//300$ parents mating, Tournament parent selection type with 20 K tournaments, scattered crossover type with 0.20 crossover probability and no mutation. We guess that we can solve this problem with $O(n\log(n))$ traces for a mutation rate of $\frac{1}{3}$ with this algorithm. We first tried with $T = n\log(n)$ traces.

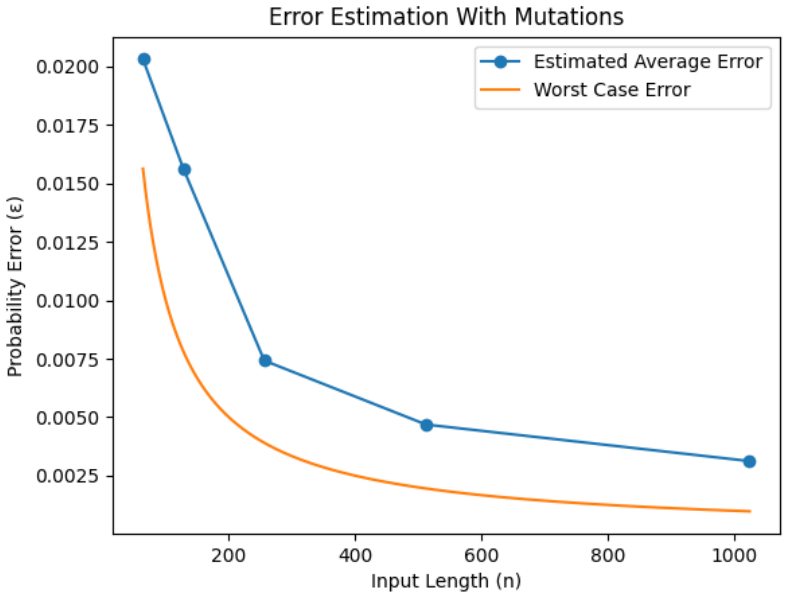


Figure 6.4: Worst case error vs average case error genetic algorithm

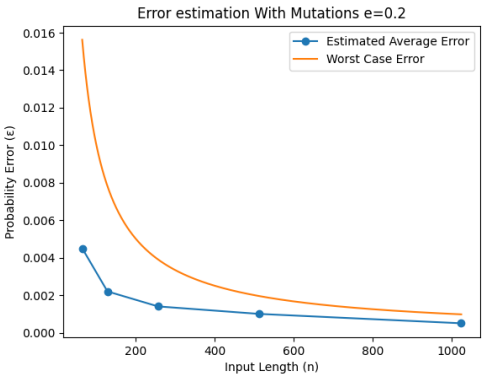
From Fig. 6.4, $n\log(n)$ traces are not sufficient to reconstruct the input string with high probability. So we guess that we will need $Cn\log(n)$ traces. Then, we increased the number of traces until $60n\log(n)$ and the average-case error was always below $1/n$. Taking into account and knowing that the number of traces and the mutation rate are inversely proportional, we set $60 = \frac{10}{0.5-\varepsilon}$. With this in mind, we plot the four plots in Fig. 6.5, varying the mutation rate.

We tried to use the genetic algorithm to analyse the TrimAndExtend and the TrimExtendAndMutate models.

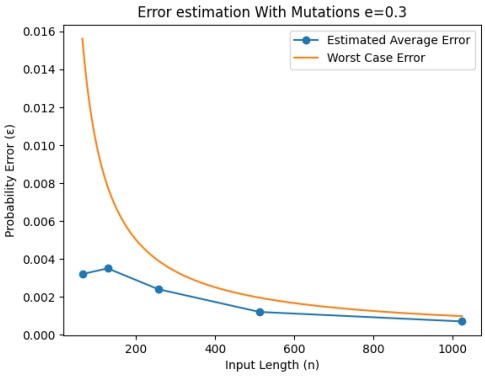
6.2 Results on TrimAndExtend model

This model is more difficult than the TrimSuffixAndExtend model. If we use the most common trace algorithm, the $\Pr(Y_x = x) = O(\frac{1}{n^2})$. The proof of this probability is the following. In our experiments, we set freely from $0 to n$ the suffix k and we conditioned

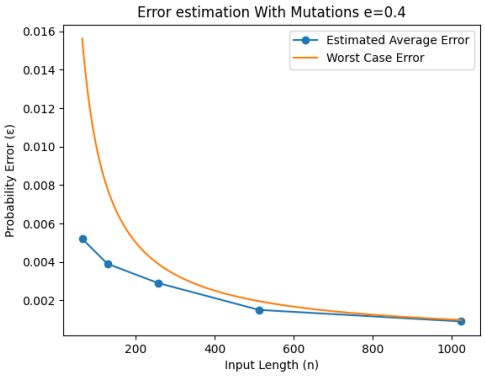
(a) Worst case error vs average case error genetic algorithm mutation rate = 0.1



(b) Worst case error vs average case error genetic algorithm mutation rate = 0.2



(c) Worst case error vs average case error genetic algorithm mutation rate = 0.3



(d) Worst case error vs average case error genetic algorithm mutation rate = 0.4

Figure 6.5: Worstcase vs average-case error for four different mutation rates.

the prefix l on the suffix. Following a similar reasoning as insection 4.1, we write

$$\Pr(Y_x = x) = \sum_{k=0}^{n} \frac{2^{-k}}{n+1} \sum_{l=0}^{n-k} \frac{2^{-l}}{n-k+1}. \tag{6.1}$$

Rearranging this equation, we say that

$$\Pr(Y_x = x) = \frac{1}{n+1} \sum_{k=0}^{n} \frac{1}{n-k+1} (2^{-(k+1)} - 2^{-n}), \tag{6.2}$$

from which we can get the upper bound

$$\Pr(Y_x = x) \leq \frac{1}{n+1} \sum_{k=0}^{n} \frac{2^{-k}}{n-k+1}. \tag{6.3}$$

To calculate this probability, we need to consider two scenarios: If $k \geq \log(n)$, then

$$\frac{2^{-k}}{n-k+1} \leq \frac{1}{n(n-\log(n)+1)}. \tag{6.4}$$

If $k \leq \log(n)$, then

$$\frac{2^{-k}}{n-k+1} \leq \frac{1}{n}. \tag{6.5}$$

Given this, we can now write

$$\Pr(Y_x = x) \leq \frac{1}{n+1}(n - \log(n))\frac{1}{n(n-\log(n))+1} + \log(n)\frac{1}{n}. \tag{6.6}$$

From here we can say that $\Pr(y_x = x) = O(\frac{1}{n^2})$, which will lead us to the upper bound $O(n^2 \log(n))$. So, we tried using a genetic algorithm to reach better sample complexity. The settings are the same as section 6.1.3, but traces are generated according to the TrimAndExtend model.

Due to the complexity of the problem, we start with $10n\log(n)$ traces. As we can

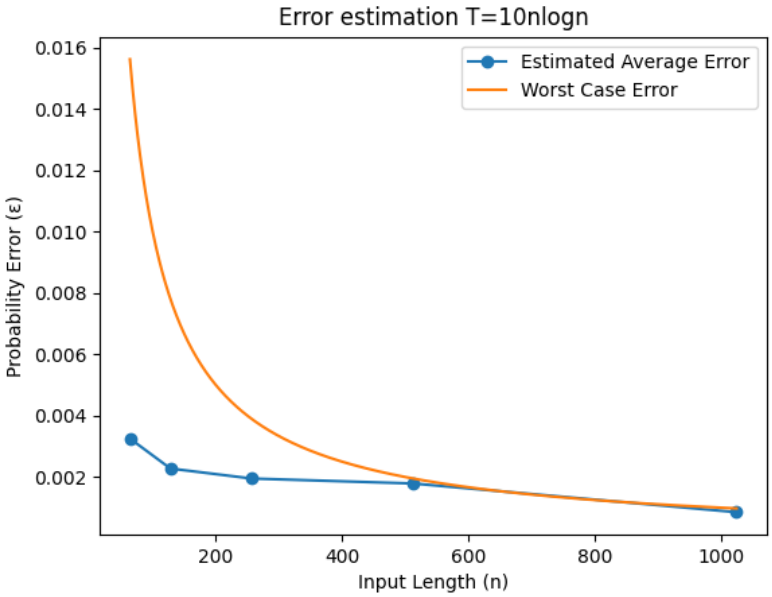


Figure 6.6: Worst-case vs Average-Case error $10n\log(n)$ traces

see in Fig. 6.6, $10n\log(n)$ traces are almost sufficient to put the average-case error below the worst-case error. So, we made another attempt with $20n\log(n)$ traces and we got the following result. Fig. 6.7 shows that $20n\log(n)$ traces are sufficient to correctly estimate the input string. So, we can assume that $Cn\log(n)$ traces are sufficient to correctly estimate the input string for the trimAndExtend model.

For the TrimExtendAndMutate model, we also tried varying the number of traces as a function of the mutation rate. We utilized $\frac{10}{0.5-\varepsilon}n\log(n)$ traces and we got the results presented in Fig. 6.8. For $\varepsilon = 0.1$ and $\varepsilon = 0.2$, the average-case error is always below the worst-case error. For $e = 0.3$ and $\varepsilon = 0.4$, the genetic algorithm seems to have some difficulty maintaining the average-case error below the average-case error.

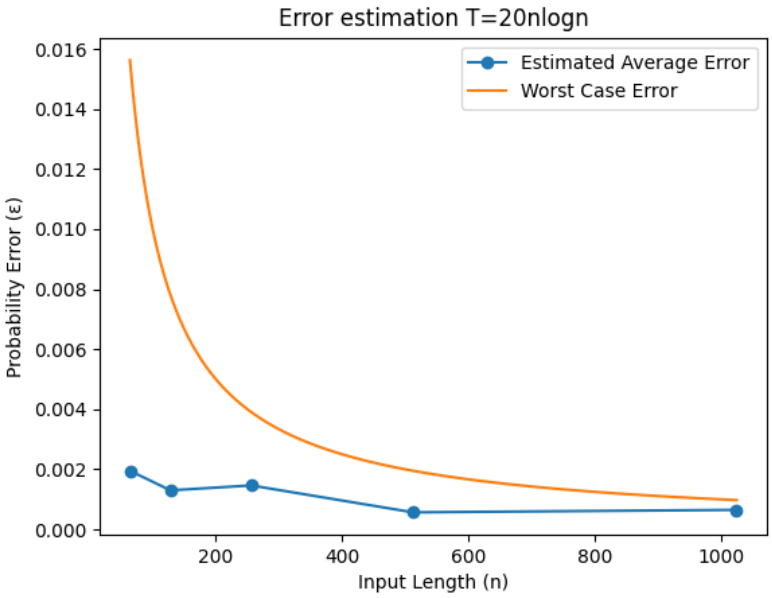
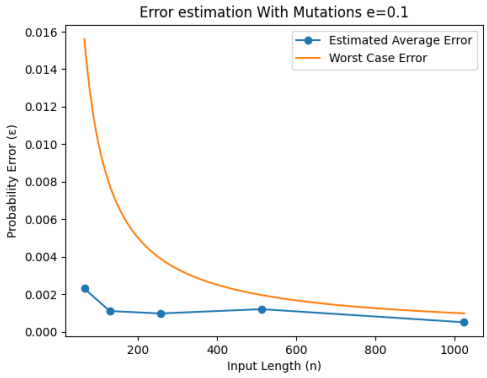
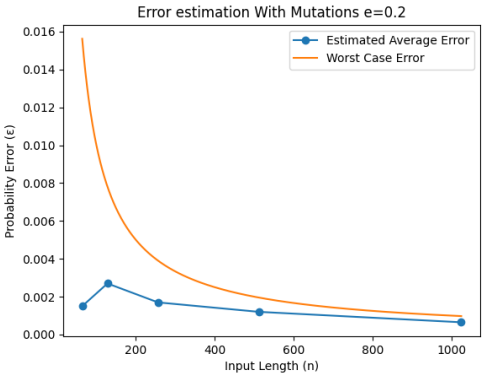


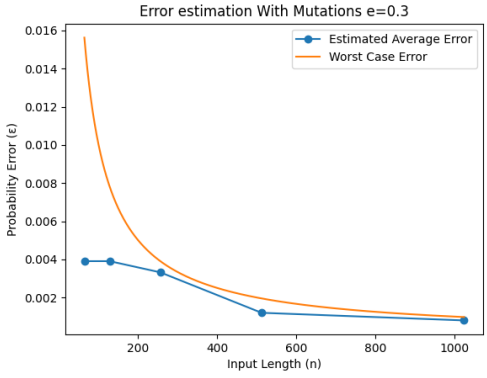
Figure 6.7: Worst-case vs Average-Case error $20n\log(n)$ traces



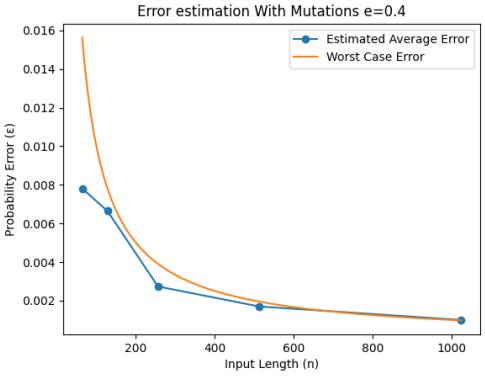
(a) Worst case error vs average case error genetic algorithm mutrate 0.1



(b) Worst case error vs average case error genetic algorithm mutrate 0.2



(c) Worst case error vs average case error genetic algorithm mutrate 0.3



(d) Worst case error vs average case error genetic algorithm mutrate 0.4

Figure 6.8: Worstcase vs average-case error for four different mutation rates.

6.3 Discussion of Empirical Results

In this section, we will discuss the results obtained in Sections 6.1 and 6.2.

Firstly, we can confidently state that for the TrimSuffixandExtend model, using a common trace algorithm with $Cn\log(n)$ traces is optimal to estimate the input x with high probability. We can assume this due to the Lower Bound deduced in Section 4.2. This lower bounds are not exclusive of the TrimSuffixAndExtend model, because, this is also applied to the TrimSuffixAndExtendWithMutations, TrimAndExtend and TrimAndExtendWithMutations. Applying the partition mean-based or the genetic algorithm here will lead to the same results. In the case of the TrimmSuffixandExtend with mutations model, the most common trace algorithm will not work well, because the probability of a trace being equal to the input string is incredibly small. So, we needed to try another approach to achieve a better result for the upper bounds. So, we used an algorithm which works in stages, achieving an upper bound of $O(n\log^2(n))$ traces. We thought that this could be the optimal upper bounds for number of traces for this model, but, after using a genetic algorithm, we could achieve $O(n\log(n))$ traces. We state that this is the near-optimal number of traces, using a genetic algorithm. Besides we have not studied the other models, we also tried to get some insights into what could be their near-optimal sample complexity.

For the TrimAndExtend model, solved with the most common trace algorithm, we need $O(n^2\log(n))$ traces. We can know this by calculating $\Pr(Y_x = x)$, where x is the input string and Y_x is a trace of x, which will give $O\left(\frac{1}{n^2}\right)$. With the genetic algorithm, we improved this value, using $O\left(n\log(n)\right)$ traces to efficiently reconstruct the input string.

7

Conclusions And Future Work

In this chapter, we will recall the objectives of this work, comparing it with the developed work and the results we have achieved. We will also discuss the importance of our findings, the challenges we have faced, as well as the future directions of our work.

7.1 Developed Work Wrap-up

At the beginning of this thesis, we proposed to design efficient algorithms to study trace reconstruction on the models described in Section 3.4, to derive the lower and upper bounds for those models. Because no one has already analysed those models, our research contributed to a kick-start in studying trace construction on them.

We successfully designed and analysed the Most Common Trace Algorithm to solve trace reconstruction on the TrimSuffixAndExtend model, deriving upper and lower bounds for this model and its worst-case probability error. We also implemented this algorithm in Python to empirically estimate the average-case error. We deduced that $O(n \log(n))$ is the optimal upper bound sample complexity for this model and $\Omega(n \log(n))$ is the optimal lower bound sample complexity.

Next, we started analysing the algorithm which process traces in stages, where we conjectured that the upper bound for the number of traces for the TrimSuffixAndExtend with mutations model is $O(n \log^2(n))$. The Lower bounds are equal to the TrimSuffixAndExtend model. To try to optimize this upper bound, we used a genetic algorithm, reaching a sample complexity of $O(n \log(n))$ traces.

Besides that we have not formally analysed an algorithm for the models TrimAndExtend and the TrimAndExtend with mutations, we used the genetic algorithm to study trace reconstruction on them, achieving upper bounds of $O(n \log(n))$ traces on both of them. We also tried to analyse the models where, after the trimming, the generated suffix and prefix have different lengths from the trimmed length, not achieving good results. This leads us to this thesis's challenges and the future directions to adopt.

7.2 Challenges and Future Directions

For the TrimSuffixAndExtend with mutations model, we only conjectured that the upper bound of the number of traces needed is $O(n \log^2(n))$ to reconstruct the input string with high probability. Continuing the analysis to state a theorem is the next step to complete the formal analysis of algorithm which processes traces in stages.

For the models TrimAndExtend and the TrimAndExtend With mutations, we only have empirical arguments with the genetic algorithm. So the next phase is to develop algorithms that can solve trace reconstruction on these models with optimal or near-optimal sample complexity, as well as their formal analysis.

The analysis of the models where the trimming and the generated suffix and prefix are different remains an open problem. Due to the synchronization problem, the mean coordinate by coordinate is difficult to apply, because the empirical mean will not be correct.

We think that a good technique to approach this problem is using deep learning, namely Recurrent Neural Networks. Due to their capacity to recognize patterns within the data, an experiment that can be done is for all traces, coordinate by coordinate, trying to estimate bit by bit the input string.

Bibliography

[1] *Alphabet - Encyclopedia of Mathematics*. https://encyclopediaofmath.org/wiki/Alphabet. (Accessed on 07/07/2023) (cit. on p. 7).

[2] T. Batu et al. "Reconstructing strings from random traces". In: *Departmental Papers (CIS)* (2004), p. 173 (cit. on pp. 11, 12, 16).

[3] V. Bhardwaj et al. "Trace reconstruction problems in computational biology". In: *IEEE Transactions on Information Theory* 67.6 (2020), pp. 3295–3314 (cit. on pp. 1, 2, 16).

[4] Z. Chase. "New lower bounds for trace reconstruction". In: (2021) (cit. on p. 15).

[5] Z. Chase. "New upper bounds for trace reconstruction". In: *arXiv preprint arXiv:2009.03296* (2020) (cit. on p. 15).

[6] X. Chen et al. *Polynomial-time trace reconstruction in the low deletion rate regime*. URL: https://doi.org/10.4230/LIPIcs.ITCS.2021.20 (cit. on p. 12).

[7] M. Cheraghchi et al. "Mean-Based Trace Reconstruction Over Oblivious Synchronization Channels". In: *IEEE Transactions on Information Theory* 68.7 (2022), pp. 4272–4281 (cit. on pp. 2, 12, 15, 16).

[8] S. I. Garcia. *L0 norm, L1 norm, L2 Norm L-Infinity Norm*. 2020-12. URL: https://montjoile.medium.com/l0-norm-l1-norm-l2-norm-l-infinity-norm-7a7d18a4f40c (cit. on p. 8).

[9] N. Holden, R. Pemantle, and Y. Peres. "Subpolynomial trace reconstruction for random strings and arbitrary deletion probability". In: *Conference On Learning Theory*. PMLR. 2018, pp. 1799–1840 (cit. on p. 15).

[10] T. Holenstein et al. "Trace reconstruction with constant deletion probability and related results". In: *Proceedings of the nineteenth annual ACM-SIAM symposium on Discrete algorithms*. 2008, pp. 389–398 (cit. on p. 12).

[11] S. Katoch, S. S. Chauhan, and V. Kumar. "A review on Genetic Algorithm: Past, present, and future". In: *Multimedia Tools and Applications* 80.5 (2020-10), pp. 8091–8126. DOI: 10.1007/s11042-020-10139-6 (cit. on pp. 21–23).

[12] A. V. Kuznetsov and A. H. Vinck. "A coding scheme for single peak-shift correction in (d, k)-constrained channels". In: *IEEE transactions on information theory* 39.4 (1993), pp. 1444–1450 (cit. on p. 1).

[13] V. I. Levenshtein. "Efficient reconstruction of sequences from their subsequences or supersequences". In: *Journal of Combinatorial Theory, Series A* 93.2 (2001), pp. 310–332 (cit. on p. 16).

[14] V. I. Levenshtein. "Reconstruction of objects from the minimum number of distorted patterns". In: *Doklady Akademii Nauk*. Vol. 354. 5. Russian Academy of Sciences. 1997, pp. 593–596 (cit. on p. 16).

[15] J. M. Lourenço. *The NOVAthesis LaTeX Template User's Manual*. NOVA University Lisbon. 2021. URL: https://github.com/joaomlourenco/novathesis/raw/main/template.pdf (cit. on p. ii).

[16] J. M. Lourenço Ribeiro. "Coding against synchronisation and related errors". In: (2021) (cit. on pp. 2, 7, 11, 12).

[17] M. Mitzenmacher and E. Upfal. *Probability and computing: Randomization and probabilistic techniques in algorithms and data analysis*. Cambridge university press, 2017 (cit. on pp. 4–6).

[18] A. Moitra and M. Saks. "A polynomial time algorithm for lossy population recovery". In: *2013 IEEE 54th Annual Symposium on Foundations of Computer Science*. IEEE. 2013, pp. 110–116 (cit. on p. 16).

[19] S. Narayanan. "Improved algorithms for population recovery from the deletion channel". In: *Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms (SODA)*. SIAM. 2021, pp. 1259–1278 (cit. on p. 16).

[20] Y. Ng et al. "Picket-shift codes for bit-patterned media recording with insertion/deletion errors". In: *IEEE Transactions on Magnetics* 46.6 (2010), pp. 2268–2271 (cit. on p. 1).

[21] R. O'Donnell. "Analysis of Boolean Functions". In: *CoRR* abs/2105.10386 (2021). arXiv: 2105.10386. URL: https://arxiv.org/abs/2105.10386 (cit. on p. 8).

[22] W. b. A. Viescinski. *Tournament selection in genetic algorithms*. 2023-12. URL: https://www.baeldung.com/cs/ga-tournament-selection (cit. on p. 22).

A

Python Code

A.1 Most Common Trace Algorithm

```
1
2  from collections import Counter
3  import numpy as np
4  from matplotlib import pyplot as plt
5
6
7
8
9  def most_common_array(arrays):
10     count = Counter([tuple(array) for array in arrays])
11     most_common = count.most_common(1)
12     return most_common[0][0]
13
14
15
16
17 def calculate_error(x, x_pred):
18     return sum([1 if x[i] != x_pred[i]
19                 else 0 for i in range(x.shape[0])]) / x.shape[0]
20
21
22
23
24
25 def generate_traces(x, T):
26     traces = []
27     for i in range(T):
28         k = int(np.random.uniform(0, n))
29         if k != 0:
30             trace = x[:-k]
31             suffix = np.random.randint(2, size=k)
32             trace = np.concatenate((trace, suffix), axis=None)
33             traces.append(trace)
34         else:
```

```
35              traces.append(x)
36      return traces
37
38
39  n_s = []
40  avg_error = []
41  worst_error_y = []
42  worst_error_x = []
43  for n in range(64, 1028, 128):
44      error = 0
45      for _ in range(300):
46          x = np.random.randint(2, size=n)
47          T = int(n * np.log2(n))
48          traces = generate_traces(x, T)
49          x_pred = np.asarray(most_common_array(traces))
50          error += calculate_error(x, x_pred)
51      avg_error.append(error / 300)
52      n_s.append(n)
53
54  for i in range(64, 1025):
55      worst_error_x.append(i)
56      worst_error_y.append(1 / i)
57
58  plt.plot(n_s, avg_error, marker="o") Most
59  plt.plot(worst_error_x, worst_error_y)
60  plt.title("Error estimation Trimmsuffix")
61  plt.legend(["Estimated Average Error", "Worst Case Error"])
62  plt.xlabel("Input Length (n)")
63  plt.ylabel("Probability Error (\u03B5)")
64  plt.savefig(f"TrimSuffixTnlogn")
65  plt.show()
```

Listing A.1: Most Common Trace Algorithm

We begin by importing the libraries to help us deal with this problem. Numpy deals with numerical calculations,matplotlib plots the graphics and collections, to count how many times a trace occurs in a set of them, returning the most common one.

The main part of the program lies between lines 39 and 52. In the beginning, we define four crucial variables: nS, representing a sequence of lengths that we want to predict for the original string; avgError, representing the estimated average error for a given length; and worstErrorX and worstErrorY, which denote the x and y values for our worst-case error, which is equal to 1/n.

Next, we iterate over different n values to create a graph that will help us evaluate the average error versus the worst error behaviour. For each n, we repeat the algorithm 300 times to get a good estimation of the average error. To do this, we generate a random binary input string x of length n for each iteration and then calculate the number of traces needed to recover x.

To create the necessary traces, we call an auxiliary function that deals with this task,

which can be found between lines 25 and 36 Firstly, we randomly generate a trimming suffix k, cut x at that suffix, generate a new suffix, and concatenate it to the cut string to build the trace. Finally, we add the trace to the list.

After generating the traces, we will return to the main program and use an auxiliary function to find the most common trace. This function can be found in lines lines 9 and 12,. It works by counting how many times each trace occurs in the list and returning the one that occurs the most.

Because the function returns a tuple, we need to convert it to a numpy array for better manipulation. Now that we have our prediction, we can compare it with the original string to calculate the error. To calculate it, we use another auxiliary function, which is between line 35 and 38. The error is simply the sum of all the positions that x and xPred differ and divide them by the length of x.

To complete the process, we first gather the error and pass it to start a new iteration. After completing 300 iterations, we calculate the average of all iterations, add it to the avgError list along with the value of n to the ns list, and then move on to the next value of n.

Once we have done this for all values of n, we compute the worst error function between 64 and 1024 and then plot the results.

This approach only works for the trimmsuffixandextend model.

A.2 Partition Mean-Based

```
1
2
3  import numpy as np
4  from matplotlib import pyplot as plt
5  from joblib import Parallel, delayed
6  import random
7
8  PACE = 0.5
9  C = 20
10 ERROR = 1 / 4
11 MUT_PROB = 0.33
12
13
14 def generate_traces(i, k, traces, n):
15     traces[i, :n - k] = x[:n - k]
16     for j in range(n - k):
17         if random.random() <= MUT_PROB:
18             traces[i, j] = 1 - traces[i, j]
19     traces[i, n - k:] = np.random.randint(2, size=k)
20     return traces
21
22
23 def mean_based( n, T):
```

```
    x = [random.choice([0, 1]) for _ in range(n)]
    x_pred = []
    start = 0
    end = n // 2
    threshold = int(C * np.log2(n))
    traces = np.zeros((T, n), dtype=int)
    k_values = np.random.randint(n + 1, size=T)
    for i, k in enumerate(k_values):
        traces = generate_traces(i, k, traces, n)

    while (end < n) or (start != n):
        if n - start <= threshold:
            end = n

        mean_y_x = np.mean(traces[:, start:end], axis=0)
        x_pred.extend((mean_y_x >= 0.5).astype(int))
        for i in range(traces.shape[0]):
            t = traces[i, start:end]
            # Find the absolute difference between the two arrays
            diff = np.abs(t - x_pred[start:end])
            # Count the number of non-zero elements in the difference array
            diff_bits = np.count_nonzero(diff)
            if diff_bits >= (MUT_PROB + ERROR) * len(x_pred[:end]):
                np.delete(traces, i, axis=0)
        start = end
        end = min(int((end + n) * 0.5), n)

    errors = sum([1 if x[i] != x_pred[i] else 0 for i in range(n)]) / n
    return errors


if __name__ == '__main__':
    n_s = []
    avg_error_n = []
    worst__error_y = []
    worst_error_x = []

    for n in range(64, 1024, 250):

        T = round(C * n * np.log2(n) ** 2)
        results = Parallel(n_jobs=14)(delayed(mean_based)
        (n, T) for _ in range(300))
        error = sum(results) / 300
        n_s.append(n)
        avg_error_n.append(error)
        print(f"{n} error: {error}")

    for i in range(64, 1025):
        worst_error_x.append(i)
        worst_error_y.append(1 / i)
```

```
74
75      plt.plot(n_s, avg_error_n, marker="o")
76      plt.plot(target_x, target_y)
77      plt.title("Error estimation With Mutations")
78      plt.legend(["Estimated Average Error", "Worst Case Error"])
79      plt.xlabel("Input Length (n)")
80      plt.ylabel("Probability Error (\u03B5)")
81      plt.savefig(f"TrimmSuffixMutation")
82      plt.show()
```

Listing A.2: Partition Mean-Based

As the algorithm described in appendix A.1 requires us to import numpy and matplotlib libraries. Additionally, we will use the joblib library to parallelize the algorithm since it involves extensive computations. We will also use the random library to generate a random number between 0 and 1. s The main part of this program is located between lines 55 and 82. Here, we define important variables and a loop to iterate over different input lengths. To improve performance, we pass meanBased to the parallel function.

Now, let's analyze the meanBased function in detail, which is located between line 135 and line 164. We first define the initial variables. The start, end, and threshold values differ from the previous algorithm because this algorithm works with partitions. Therefore, we need to set the beginning and end of the part of x that we want to recover. The threshold value is used to indicate when we should apply the mean when only a few bits need to be recovered.

In this case, the traces are generated in a similar way as in appendix A.1. But, after the trimming is done, for all the reaming bits, we sample a number between 0 and 1 and if it is below the mutation probability, we flip the bit.

We are now able to explain how the algorithm works, as outlined in line 23 and. To begin, we analyze the traces between 0 and $\frac{n}{2}$. We calculate the mean of this window for all of these traces, which serves as our protection for the first half of our variable x. We then add it to the xPred variable.

During the process, we compare each trace with the prediction and identify the different bit positions. If the difference between them exceeds the limit specified in line 46, we eliminate the trace. This process continues until the limit specified in line 35 is reached. Once this limit is reached, we apply the mean to the remaining traces and return the prediction. Finally, we compare each bit of the prediction with the original string to find the associated error.

After getting the errors, we compute the average and keep both the average error and input length. Then we move to the next iteration. In the end, we draw the worst-case error and generate the plots, as we did before.

A.3 Genetic Algorithm

```python
import numpy as np
from joblib import Parallel, delayed
from numpy.random import rand
import pygad
from random import randint
from matplotlib import pyplot as plt

# average of population, must global in order to update it



# Fitness function
def fitness_function(ga, solution, ix_solution):
    return sum([1 if solution[i] == avg_pop[i] else 0 for i in range(len(
    solution))]) / len(solution)


# generate population
def generate_population(x, T, e):
    pop = []
    k_values = np.random.randint(len(x) + 1, size=T)
    for k in k_values:
        if k != 0:
            individual = x[:-k]
            for i in range(len(individual)):
                if rand() < e:
                    individual[i] = 1 - individual[i]
            suff = [randint(0, 1) for _ in range(k)]
            individual.extend(suff)
            pop.append(individual)
        else:
            individual = x.copy()
            for i in range(len(x)):
                if rand() < e:
                    individual[i] = 1 - individual[i]
            pop.append(individual)
    return pop


def callback_function(ga):
    verbose = 2
    last_fitness = 0
    atual_pop = ga.population
    avg_pop = np.mean(atual_pop, axis=0)
    avg_pop = (avg_pop >= 0.5).astype(int)

    if ga.generations_completed % verbose == 0:
        print("Generation = {generation}".format(generation=ga.
    generations_completed))
        print("Fitness    = {fitness}".format(fitness=ga.best_solution()[1]))
```

```
49
50
51 # genetic algorithm
52
53 def genetic_algorithm(n, T, e):
54     x = [randint(0, 1) for _ in range(n)]
55     initial_population = generate_population(x, T, e)
56     global avg_pop
57
58     avg_pop = np.mean(initial_population, axis=0)
59     avg_pop = (avg_pop >= 0.5).astype(int)
60
61     ga = pygad.GA(num_generations=20,
62                   initial_population=initial_population,
63                   fitness_func=fitness_function,
64                   on_generation=callback_function,
65                   num_parents_mating=T // 300,
66                   parent_selection_type="tournament",
67                   K_tournament=20,
68                   crossover_type="scattered",
69                   crossover_probability=0.20,
70                   mutation_type="random",
71                   mutation_probability=0.00,
72                   gene_type=int,
73                   )
74
75     ga.run()
76     avg_pop = np.mean(ga.population, axis=0)
77     avg_pop = (avg_pop >= 0.5).astype(int)
78     error = sum([1 if x[i] != avg_pop[i] else 0 for i in range(len(x))]) / len
       (x)
79     print(f"individual error {error}")
80     return error
81
82
83 # Let's start the algorithm
84 if __name__ == '__main__':
85
86     n_s = []
87     errors = np.zeros(shape=(5, 4))
88     target_x = []
89     target_y = []
90     n = 64
91     i = 0
92     while n < 1025:
93         for e in np.arange(0.1, 0.5, 0.1):
94             j = 0
95
96             T = int(10 / (0.5 - e) * n * np.log2(n))
97             error = Parallel(n_jobs=-1)(
```

```
98                delayed(genetic_algorithm)(n, T, e) for _ in range(300
99                                                                    ))
100           error = sum(error) / 300
101           print(f"The average error for n = {n} and error = {e} is {error}")
102           errors[i, j] = e
103           j += 1
104       i += 1
105       n_s.append(n)
106       n *= 2
107
108   for i in range(64, 1025):
109       target_x.append(i)
110       target_y.append(1 / i)
111
112   e = 0.1
113   for i in range(4):
114       e = 0.1
115       plt.plot(n_s, errors[i,:], marker="o")
116       plt.plot(target_x, target_y)
117       plt.title(f"Error estimation With Mutations e={e}")
118       plt.legend(["Estimated Average Error", "Worst Case Error"])
119       plt.xlabel("Input Length (n)")
120       plt.ylabel("Probability Error (\u03B5)")
121       plt.savefig(f"GeneticAlgorithme={i}")
122       plt.show()
123       e+=0.1
```

The code above implements a genetic algorithm to maximize the population's average and estimate the original input string. To implement the genetic algorithm, we need to import libraries that will help us with this task. The Pygad library is the most efficient library for developing genetic algorithms, and we have imported it for this purpose. We believe that this is the optimal algorithm for the trimsuffixandExtend with mutations model. To find the best trade-off between the number of traces and the mutation error, we vary different mutation errors and run several input lengths.

We have empirically established that 20 generations are enough to converge to a good result. The trace generation follows the model described in **??**. The code block between lines 53 and 80 is the important part.

First, we generate the initial population, and then its average is calculated coordinate by coordinate. After this, we instantiate our genetic algorithm and fill in a set of parameters to train our algorithm to better optimize the population average. The initial population consists of the traces that we generate. The fitness function, which is in line 13, compares the actual average population with each trace. The ones that are far from the average will be less fit than the others.

The on-generation parameter indicates what is done after the conclusion of a generation. In our case, we update the average with the new optimized population and then print some statistics. The numparentsmating is the number of solutions selected as parents to

generate the next population. We select the tournament selection type (see section 3.5.3 for a more detailed explanation) and perform tournaments between 20 traces, specified by the variable Ktournament.

We chose the scattered crossover type (see section 3.5.4 for more details) with a probability of 0.20. This rate is low because we do not need a lot of changes on the traces, only a few variations in the population. Setting this rate too high would introduce a lot of diversity, compromising the convergence of the average to the input string.

Finally, we do not introduce any type of mutation since introducing more errors will lead to a worse estimation. After running the algorithm, we compare the average after 20 generations with the input string and return the error. The rest of the code is similar to the other two: running several times, computing the average error, and then plotting it against the worst-case error in a graphic.

This is the genetic algorithm version for mutation models. For the models without mutations, the population generation is similar, expecting we do not mutate any during the population generation process and not varying the number of traces as a function of the mutation error. We implemented, also, this algorithm for working with suffix and prefix trimming. To do this, we substitute the function in line 18 for

```
1
2 pop = []
3     k_values = np.random.randint(len(x), size=T)
4
5     for k in k_values:
6         if k != 0:
7             l = np.random.randint(len(x) - k)
8             individual = x[l:-k]
9             suff = [randint(0, 1) for _ in range(k)]
10            pref = [randint(0, 1) for _ in range(l)]
11            pref.extend(individual)
12            pref.extend(suff)
13            for ix,b in enumerate(pref):
14                if random.random() <= MUT_RATE:
15                    pref[ix] = 1 - pref[ix]
16            pop.append(pref)
17        else:
18            individual = x.copy()
19            l = np.random.randint(len(x) + 1)
20            individual = individual[l:]
21            pref = [randint(0,1) for _  in range(l)]
22            pref.extend(individual)
23            for ix, b in enumerate(pref):
24                if random.random() <= MUT_RATE:
25                    pref[ix] = 1 - pref[ix]
26            pop.append(pref)
```

In addition to generating a suffix, we generate also a prefix.