Reconstruction Algorithms for DNA-based Storage Systems

Omer Sabary
Reconstruction Algorithms for DNA-based Storage Systems

Research Thesis

Submitted in partial fulfillment of the requirements for the degree of Master of Science in Computer Science

Omer Sabary

Submitted to the Senate of the Technion — Israel Institute of Technology
Sivan 5780 Haifa June 2020
The research thesis was done under the supervision of Prof. Eitan Yaakobi in the Computer Science Department.

Acknowledgments

I would like to thank my advisor, Prof. Eitan Yaakobi for guiding, supporting me, and believing in me over the years. You have set an example of excellence as a researcher, mentor, instructor, and role model.

The generous financial support of the Technion is gratefully acknowledged.
# Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract</td>
<td>1</td>
</tr>
<tr>
<td>Abbreviations and Notations</td>
<td>3</td>
</tr>
<tr>
<td><strong>1 Introduction</strong></td>
<td>4</td>
</tr>
<tr>
<td>1.1 Motivation</td>
<td>4</td>
</tr>
<tr>
<td>1.2 Our Work</td>
<td>6</td>
</tr>
<tr>
<td>1.3 Outline of the Thesis</td>
<td>7</td>
</tr>
<tr>
<td><strong>2 Background</strong></td>
<td>8</td>
</tr>
<tr>
<td>2.1 DNA Basics</td>
<td>8</td>
</tr>
<tr>
<td>2.2 DNA Storage Systems</td>
<td>9</td>
</tr>
<tr>
<td><strong>3 SOLQC: Synthetic Oligo Library Quality Control Tool</strong></td>
<td>10</td>
</tr>
<tr>
<td>3.1 Errors in DNA Storage Systems</td>
<td>10</td>
</tr>
<tr>
<td>3.2 Related Work</td>
<td>11</td>
</tr>
<tr>
<td>3.3 SOLQC Tool</td>
<td>12</td>
</tr>
<tr>
<td>3.4 Statistical QC Analysis for Synthetic DNA Libraries</td>
<td>15</td>
</tr>
<tr>
<td>3.4.1 Composition statistics</td>
<td>15</td>
</tr>
<tr>
<td>3.4.2 Error statistics</td>
<td>20</td>
</tr>
<tr>
<td>3.5 Results</td>
<td>25</td>
</tr>
<tr>
<td>3.5.1 Pre-processing and Filtering of the Libraries</td>
<td>25</td>
</tr>
<tr>
<td>3.5.2 Analysis of Synthetic DNA Datasets</td>
<td>26</td>
</tr>
<tr>
<td>3.6 Use-Case Examples</td>
<td>28</td>
</tr>
<tr>
<td>3.7 Installation</td>
<td>29</td>
</tr>
<tr>
<td>3.7.1 Installing Docker</td>
<td>29</td>
</tr>
<tr>
<td>3.7.2 Installing SOLQC</td>
<td>30</td>
</tr>
</tbody>
</table>
# The Error Probability of Maximum Likelihood Decoding over Two Deletion/Insertion Channels  
## 4.1 Definitions and Preliminaries  
## 4.2 The Deletion and Insertion Channels  
## 4.3 Two Deletion Channels  
## 4.4 Two Insertion Channels  

# Reconstruction Algorithms for DNA-Storage Systems  
## 5.1 Preliminaries and Problem Definition  
### 5.1.1 The DNA Reconstruction Problem  
## 5.2 Related Work  
### 5.2.1 Reconstruction Algorithms for DNA-Storage Systems  
### 5.2.2 Theoretical Results on the Trace Reconstruction Problem  
## 5.3 The Deletion DNA Reconstruction Problem  
### 5.3.1 An Algorithm for Small Fixed Values of $t$  
### 5.3.2 Simulations  
### 5.3.3 Large Cluster  
## 5.4 The DNA Reconstruction Problem  
### 5.4.1 The LCS-Anchor Algorithm  
### 5.4.2 The Iterative Reconstruction Algorithm  
### 5.4.3 The Divider BMA Algorithm  
## 5.5 Results  
### 5.5.1 Results on Simulated Data  
### 5.5.2 Results on Data from DNA Storage Experiments  

# Conclusions and Future Work  
## 6.1 Summary  
## 6.2 Future Work
List of Figures

1.1 A DNA-based storage system ........................................ 4

3.1 An example of 25 reads (in purple) aligned to a variant of length 27 (in yellow). For each read, the locations of the deletions, substitutions, insertions are marked in red, blue, green, respectively. This alignment output forms the basis of the analysis performed by SOLQC .................................................. 14

3.2 Base distribution in the design variants (see 3.4.1.1) ............. 16

3.3 Base distribution in the reads (see 3.4.1.1) .......................... 16

3.4 Histogram of the number of filtered reads per variant (see 3.4.1.2) ............................................................. 17

3.5 Histogram of the number of filtered reads per variant, stratified by the GC-content (see 3.4.1.2) ................................. 18

3.6 Sorted bar plot of the number of filtered reads per variant (see 3.4.1.3) ............................................................. 19

3.7 Sorted bar plot of the number of filtered reads per variant, stratified by the GC-content (see 2.2.1.3) ............................. 19

3.8 Histogram of the length of reads (see 3.4.1.4). Note that the difference between 152 and 154 is two orders of magnitude ............. 20

3.9 Total error rates. (see 3.4.2.1) ......................................... 21

3.10 Error rates stratified by symbol. Note that the numbers are in percents. For example the value of 0.024 for ”A” long deletion, means that 0.024 percents of the occurrences of base A in the library creates long deletion error. (see 3.4.2.2) .......................... 22

3.11 Cumulative distribution based upon the number of errors (see 3.4.2.3). Note that 70% of the reads have neither sequencing nor synthesis error .................................................. 22

3.12 Deletion length distribution (see 3.4.2.4) ........................... 23
3.13 Error rates by position (see 3.4.2.5). X-axis represents position counted from the 5’ end of the designed variant. The Y-axis is log-scale. ................................. 24

3.14 Error rates by the GC-content (see 3.4.2.6). ................. 24

3.15 Total error rates in the four datasets. Note that total error rates are also due to sequencing errors. While EZ-17 [22], G-15 [27], and O-17 [48] were sequenced on Illumina sequencing machines, the fourth dataset, Y-16 [70], used a much noisier sequencing platform, which explains the differences in the error rates. ............................... 31

3.16 Cumulative distributions based upon the number of errors. Note differences in the range of Y-axes in the four figure. Also note that the X-axes are truncated (see 3.4.2.3). ................................. 32

3.17 Error rates, stratified by symbol in the four datasets (see 3.4.2.2). Note that the colorbars are different in each plot. ................. 33

3.18 Histograms of the length of the reads in EZ-17 [22], using different filtering schemes (1), (2), and (3), respectively (see 3.4.1.4). ................................. 34

3.19 Histograms of the cluster size per variant in two of the four datasets (see 3.4.1.2). ................................. 35

3.20 Error rates by the GC-content in the four datasets. Note the differences in the range of the Y-axis in the four figures (see 3.4.2.6). ................. 36

3.21 Error rates per position in the four datasets. X-axis represents position counted from the 5’ end of the designed variant, note the differences in the range of the X-axis and Y-axis in the four figures (see 3.4.2.5). ................................. 37

4.1 Levenshtein error rate by the deletion probability \( p \). These simulations verify Theorem 4.8. ................................. 50

4.2 Levenshtein error rate by the deletion probability \( p \). These simulations verify Lemma 4.5 and Lemma 4.6. ................................. 50

4.3 Failure rate of the ML decoder, stratified by the coding scheme. These simulations verify Theorem 4.9 and Theorem 4.10. ................................. 51

4.4 Levenshtein error rate by the insertion probability \( p \). This simulation verifies Theorem 4.14. ................................. 54
5.1 Levenshtein error rate by the deletion probability \( p \), for clusters of size 7 (left) and 10 (right). This figure presents results from Algorithm 1, the BMA algorithm, and the \( p^t \) lower bound. Note that the LER was 0 for \( p \leq 0.06 \) and \( p \leq 0.03 \) for \( t = 10 \) and \( t = 7 \), respectively. The X-axis represents the different values of the deletion probability in the range \([0.01, 0.20]\) and the Y-axis represents the average LER of the clusters.

5.2 \( k \)-error success rate, \( k_{1,succ} \) and \( k_{j,succ} \) values by the cluster size \( 4 \leq t \leq 10 \). The deletion probability was 0.25.

5.3 \( k \)-mer distance demonstration for 4 traces. The original strand \( x \) is of length \( n = 20 \).

5.4 Performance evaluation of Algorithm 1 and Algorithm 3. The simulation were for clusters of size \( t = 100 \), design length of \( n = 200 \), for each probability \( p \) we simulate 10,000 clusters. In all of the simulations the original sequence was reconstructed by the algorithms, introducing a success rate of 1.

5.5 Example input to Algorithm 1.

5.6 Algorithm 1 Example - Patterns of \( y_1 \).

5.7 Algorithm 1 Example - Patterns of \( y_1 \), their indices and their frequencies. The numbers on the left side of each list are the indices. The numbers on the right side of each pattern in a list represents the pattern’s frequency.

5.8 Edit error rate by the error probabilities for the cluster sizes \( t = 10 \) and \( t = 20 \). The length of the original sequence was \( n = 100 \) and the error probabilities ranges between 0.029701 and 0.271.

5.9 Success rate by the error probabilities for the cluster sizes \( t = 10 \) and \( t = 20 \). The length of the original sequence was \( n = 100 \) and the error probabilities ranges between 0.029701 and 0.271.

5.10 \( k_{1,succ} \) values by the error probabilities for the cluster sizes \( t = 10 \) and \( t = 20 \). The length of the original sequence was \( n = 100 \) and the error probabilities ranges between 0.029701 and 0.271.

5.11 Edit error rate by the reconstruction algorithm, for data from DNA storage experiments [22, 27, 48].
List of Tables

3.1 Experiment by Erlich and Zielinski [22] .......................... 15
3.2 Synthetic DNA Libraries ............................................... 25
Abstract

The trace reconstruction problem was first proposed in [5]. Under this framework, a length-$n$ string $x$, yields a collection of noisy copies, called traces, $y_1, \ldots, y_t$, where each $y_i$ is independently obtained from $x$ by passing through a deletion channel, which deletes every symbol with some fixed probability $p_d$. Suppose the input string $x$ is arbitrary. In the trace reconstruction problem, the main goal is to determine the required minimum number of i.i.d traces in order to reconstruct $x$ with high probability. The trace reconstruction problem can be extended to the model where each trace is a result of $x$ passing through a deletion-insertion-substitution channel. Here, in addition to deletions, each symbol can be switched with some substitution probability $p_s$, and for each $j$, with probability $p_i$, a symbol is inserted before the $j$-th symbol of $x$.

Motivated by the storage channel of DNA [28, 48, 70], this work is focused on another variation of the trace reconstruction problem, which is referred by the DNA reconstruction problem. The setup is similar to the trace reconstruction problem. A length-$n$ string $x$ is transmitted $t$ times over the deletion-insertion-substitution channel and generates $t$ traces $y_1, \ldots, y_t$. A DNA reconstruction algorithm is a mapping $R : (\Sigma_q)^t \rightarrow \Sigma_q$ which receives the $t$ traces $y_1, \ldots, y_t$ as an input and produces $\hat{x}$, an estimation of $x$. The goal in the DNA reconstruction problem is to minimize $d_e(x, \hat{x})$, i.e., the edit distance between the original string and the algorithm’s estimation.

This work consists of three parts. The first part presents a software tool to support quality control of synthetic DNA libraries. This part includes a comprehensive study and characterization of the errors in the DNA storage channel.

The second part studies the maximum likelihood (ML) decoder for two deletion channels. In this part, we calculate the error and the failure probability of the ML decoder, and also simulate it to verify our results.

In the third part we present several new algorithms for the DNA reconstruc-
tion problem and for the deletion DNA reconstruction problem. While most of the previous algorithms look locally on each symbol and use a symbol-wise majority technique, our algorithms look globally on the entire sequence of the traces and use dynamic programming algorithms in order to estimate the original sequence. This global approach builds upon the algorithms to find the shortest common supersequence and the longest common subsequence of a set of given sequences. Our algorithms also differ from previous results by introducing less limitations on the input, and more than that, they perform well even for error probabilities as high as 0.27. Lastly, we tested our algorithms on simulated data and on data from previous DNA experiments [22, 27, 48]. We compared the edit error rates of our algorithms with the edit error rates of previous algorithms [5, 26, 66]. In all of the tests our algorithms presented significant improvements of the edit error rates.
## Abbreviations and Notations

- \( k \in \mathbb{Z}, \{1, \ldots, k\} \)
- \( \Sigma_q \) — \( q \)-ary alphabet, \( \{0, \ldots, q - 1\} \)
- \( x \in \Sigma_q^n \) — \( (x_1, \ldots, x_n), x_i \in \Sigma_q \)
- \( |x| \) — The length of \( x \in \Sigma_q^n \)
- \( d_H(x, y) \) — The Hamming distance between \( x \) and \( y \)
- \( d_L(x, y) \) — The Levenshtein distance between \( x \) and \( y \)
- \( d_e(x, y) \) — The edit distance between \( x \) and \( y \)
- \( SCS(y_1, \ldots, y_t) \) — The length of the shortest common supersequence (SCS) of \( y_1, \ldots, y_t \)
- \( LCS(y_1, \ldots, y_t) \) — The length of the longest common subsequence (LCS) of \( y_1, \ldots, y_t \)
- \( \log x \) — \( \log_2 x \)
Chapter 1

Introduction

1.1 Motivation

Recent studies presented a significant progress in DNA synthesis and sequencing technologies \([4, 13, 56, 38, 49, 58]\). This progress also introduced the development of data storage technology based upon DNA molecules. A DNA storage system consists of three important components. The first is the DNA synthesis which produces the oligonucleotides, also called strands, that encode the data. In order to produce strands with acceptable error rates, in a high throughput manner, the length of the strands is typically limited to no more than 250 nucleotides \([6]\). The second part is a storage container with compartments which stores the DNA strands, however without order. Finally, sequencing is performed to read back a representation of the strands, which are called reads.

Current synthesis technologies are not able to generate a single copy for each DNA strand, but only multiple copies where the number of copies is in the order of thousands to millions. Moreover, sequencing of DNA strands is usually preceded
by PCR amplification which replicates the strands. Hence, every strand has multiple copies and several of them are read during sequencing.

The encoding and decoding stages are two processes, external to the storage system, that convert the user’s binary data into strands of DNA such that, even in the presence of errors, it will be possible to revert back and recover the original binary data. The decoding procedure consists of three steps, which we refer by 1. clustering, 2. reconstruction, and finally 3. error correction. After the strands are read back by sequencing, the first task is to partition them into clusters such that all strands in the same cluster originated from the same synthesized strand. After the clustering step, the goal is to reconstruct each strand based upon all its noisy copies, and this stage is the main problem studied in this thesis. Lastly, errors which were not corrected by the reconstruction step, mis-clustering errors, lost strands, and any other error mechanisms should be corrected by the use of an error-correcting code.

Any reconstruction algorithm for the second stage is performed on each cluster to recover the original strand from the noisy copies in the cluster. Having several copies for each strand is beneficial since it allows to correct errors that may occur during this process. In fact, this setup falls under the general framework of the string reconstruction problem which refers to recovering a string based upon several noisy copies of it. Examples for this problem are the sequence reconstruction problem which was first studied by Levenshtein and the trace reconstruction problem. In general, these models assume that the information is transmitted over multiple channels, and the decoder, which observes all channel estimations, uses this inherited redundancy in order to correct the errors.

Generally speaking, the main problem studied under the paradigm of the sequence reconstruction and trace reconstruction problems is to find the minimum number of channels that guarantee successful decoding either in the worst case or with high probability. However, in DNA-based storage systems we do not necessarily have control on the number of strands in each cluster. Hence, the goal of this work is to propose efficient algorithms for the reconstruction problem as it is reflected in DNA-based storage systems where the cluster size is a given parameter. Then, the goal is to output a strand that is close to the original one so that the number of errors the error-correcting code should correct will be minimized. We will present algorithms that work with a flexible number of copies and various probabilities for deletion, insertion, and substitution errors.

In our model we assume that the clustering step has been done successfully.
This could be achieved by the use of indices in the strands and other advanced coding techniques; for more details see [57] and references therein. Thus, the input to the algorithms is a cluster of noisy read strands, and the goal is to efficiently output the original strand or a close estimation to it with high probability.

This thesis consists of three parts. In the first part we built a quality control tool to characterize the errors and their behavior of synthetic DNA libraries, and more specifically DNA-storage. In the second part, since deletion is the most dominant error in the DNA-storage channel, we studied the maximum likelihood decoder as a reconstruction algorithm for two deletion channels and calculated its error probability. In the last part, we defined and presented the DNA-reconstruction problem, the reconstruction problem as it reflected in DNA-storage systems and designed algorithms to solve it. We also evaluate the accuracy of our algorithms and compare them to other previously published algorithms.

1.2 Our Work

In this work we present several reconstruction algorithms, designed for DNA storage systems. Since our purpose is to solve the reconstruction problem as it is reflected in DNA-storage systems, our algorithms aim to minimize the distance between the output and the original designed strands. The algorithms in this work are different from most of the previously published reconstruction algorithms in several respects. First, we do not require any assumption on the input. That is, the input can be arbitrary and does not necessarily belong to an error-correcting code. Second, our algorithms are not limited to specific cluster size, do not require any dependencies between the error probabilities, and do not assume zero errors in any specific location of the strands. Third, we can limit the complexity of our algorithms, so they can run with practical time on actual data from previous DNA-storage experiments. Lastly, since clusters in DNA storage systems may vary in their size and errors distributions, our algorithms are not limited to specific cluster size or error rate. Moreover, our algorithms are designed to minimize the distance between our output and the original strand, taking into account these errors can be corrected by the use of an error-correcting code.
1.3 Outline of the Thesis

The rest of the thesis is organized as follows. In Chapter 2, we introduce the basic terms related to DNA data storage and review some of the DNA storage systems presented recently to understand the motivation for this study. In Chapter 3, we present the software tool we created for quality control of synthetic DNA libraries. We also present results of error characterization based on data from previously published DNA storage experiments. In chapter 4, we present our calculation of the error and failure probability of the maximum likelihood decoder over two deletion channels and two insertion channels. In Chapter 5, we present reconstruction algorithms from previous work, and other related results regarding the reconstruction problem. Then, we present our reconstruction algorithms for the DNA storage systems and present a comparison of them with other algorithms. Lastly, Chapter 6 concludes and summarizes the results of this work.
Chapter 2

Background

2.1 DNA Basics

Single stranded *Deoxyribonucleic Acid* (DNA) is a molecule constructed as a linear sequence, which consists of four types of nucleotides: Adenine (A), Cytosine (C), Guanine (G), and Thymine (T). A nucleotide in a strand is also referred to as a base. Motivated by the chemical bonds that can form between the different nucleotides, A and T are said to be the complement bases of each other, and likewise for G and C. A single DNA strand, also called an *oligonucleotide* (oligo), is an ordered sequence of some combination of these nucleotides. The beginning and end of a DNA strand are distinguishable as they are chemically different. Each strand starts with the so-called 5’ nucleotide end and it ends with the 3’ nucleotide end. Single DNA strands can be synthesized chemically and modern DNA synthesizers can string together the four DNA nucleotides to form almost any possible sequence. This process enables to store digital data in the strands. The data can be read back with common DNA sequencers. The most popular sequencing technologies use DNA polymerase enzymes and are referred to as *sequencing by synthesis*.

**DNA Synthesis** is the process of creating DNA molecules. Arbitrary single stranded DNA sequences of length few hundreds bases can be generated chemically. This process results in a DNA pool which consists of all the synthesized strands. In the context of storage, synthesis corresponds to the writing channel of storing data in the DNA storage system. However, this process does not permit structured addressing of the written data.

**DNA Sequencing** is the process of reading single stranded DNA sequences from the DNA pool. The most popular technique, called *sequencing by synthesis*, involves an enzyme and free fluorescent nucleotides. There is one fluorescent
nucleotide per each of the letters, corresponding to A, C, G, and T, and having different fluorescence colors. The enzyme helps complementary nucleotides to form bonds and as a result, the original single strand in the pool binds nucleotide after nucleotide, with the free fluorescent nucleotides to form a double stranded sequence. Since each type of fluorescent nucleotide emits a different color in the binding process, it is possible to read the complement strand optically.

2.2 DNA Storage Systems

One of the early experiments of data storage in DNA was conducted by Clellan et al. in 1999. In their study they coded and recovered a message consisting of 23 characters [17]. Three sequences of nine bits each, have been successfully stored by Leier et al. in 2000. Gibson et al. [24] presented in 2010 a more significant progress, in terms of the amount of data stored successfully. They demonstrated in-vivo storage of 1,280 characters in a bacterial genome. The first large scale demonstrations of the potential of in vitro DNA storage were reported by Church et al. who recovered 643 KB of data [16] and by Goldman et al. who accomplished the same task for a 739 KB message [25]. However, both of these pioneering groups did not recover the entire message successfully and no error correcting codes were used. Shortly later, in [27], Grass et al. have managed to successfully store and recover a 81 KB message, in an encapsulated media, and Bornholt et al. demonstrated storing a 42 KB message [8]. A significant improvement in volume was reported in [7] by Blawat et al. who successfully stored 22 MB of data. Erlich and Zielinski improved the storage density and stored 2.11 MB of data [22].

The largest volume of stored data was reported by Organick et al. in [48] who stored roughly 200 MB of data, an order of magnitude more data than previously reported. Yazdi et al. developed in [71] a method that offers both random access and rewritable storage and in [70] a portable DNA-based storage system. Recently, Anavy et al. [1] enhanced the capacity of the DNA storage channel by using composite DNA letters. A similar approach, on a smaller scale, was reported in [15]. Lopez et al. stored and decoded a 1.67 MB of data in [43]. In their work they focused on increasing the throughput of nanopore sequencing by assembling and sequencing together fragments of 24 short DNA strands. Recent studies also presented an end-to-end demonstration of DNA storage [63], the use of LDPC codes for DNA-based storage [11], a computer systems perspective on molecular processing and storage [10], and lastly, the work of Tabatabaei et al. [62] which uses existing DNA strands as punch cards to store information.
Chapter 3

SOLQC: Synthetic Oligo Library Quality Control Tool

In this chapter we describe SOLQC, a software tool that supports the statistical analysis and quality control of OLs. The tool is designed to enable and to facilitate individual labs obtaining information about DNA libraries and performing error analysis before or during experiments. We describe our methods and demonstrate the results of analyzing several libraries from the literature.

3.1 Errors in DNA Storage Systems

The processes of synthesizing, storing, sequencing, and handling strands are all error prone. Each step in these processes can independently introduce a significant number of errors. Additionally, the DNA storage channel has several attributes which distinguish it from other storage media such as tapes, hard disk drives, and flash memories. We summarize some of these differences and the special error behavior in DNA.

1. Both the synthesis and sequencing processes can introduce deletion, insertion, and substitution errors on each of the read and synthesized strands.

2. Current synthesis methods can not generate one copy for each design strand. They all generate thousands to millions of noisy copies, while different copies may have a different error distribution. Moreover, some strands may have a
significant larger number of copies, while some other strands may not have copies at all.

3. The use of DNA for storage or other applications typically involves PCR amplification of the strands in the DNA pool \cite{28}. PCR is known to have a preference for some strands over others, which may further distort the distribution of the number of copies of individual strands and their error profiles \cite{50,52}.

4. Longer DNA strands can be sequenced using designated sequencing technologies, e.g. PacBio and Oxford Nanopore \cite{11,43,48,70}. However, the error rates of these technologies can be significantly higher and can grow up to 30%, with deletions and substitutions as the most dominant errors \cite{53}.

3.2 Related Work

Most of the research on characterizing errors in synthetic DNA libraries has been done in the context of individual studies using synthetic DNA. Tian et al. showed in \cite{65} that the rate of deletion is $1/100$ per position, insertion is $1/400$ per position, and the rate for substitution is $1/400$. Later, Kosuri and Church \cite{36} noted that column-based oligo synthesis has total error rate of approximately $1/200$ or less for oligos of 200 bases, where the most dominant error is a single base deletion. In addition, they showed that high GC content, at more than 50% of the bases in the strand being G or C, can inhibit the assembly and lead to lost data. They also pointed out that in OL synthesis, a synthesis method based on DNA microarrays, the error rates are usually higher than those for column-based synthesis. Recently, in \cite{28}, Heckel, Mikutis, and Grass, studied the errors in a DNA storage channel based upon three different data sets from the experiments in \cite{22,25,28}. In their work they studied the deletion/insertion/substitution rate and how it is affected by filtering reads with incorrect length (compared to the designed length). In particular, when they considered only reads with the correct length, they showed, as expected, that the deletion rate has been significantly decreased in all of the data sets. They also investigated the conditional error probability for substitutions and found out that in \cite{22} the most dominant substitution error was from G to T (20%), and in the rest of the experiments, the most dominant substitution error was from C to G (about 30-40%). They also examined the effect of the number of PCR cycles on the coverage depth, which is the distribution of the number of reads per each of
the variants. They concluded that, since the efficiency of the PCR amplification on each of the strands is different, a larger number of PCR amplification cycles leads to a higher differences in the coverage depth distribution of the variants. Organick et al. also characterized the errors in their experiment [48]. First, they found that substitution was the most frequent error in the library, then deletion, and lastly insertion. Furthermore, they found that while deletions showed almost equal rates for all of the four bases, insertions were mostly associated with base G, and substitutions were mostly associated with base T. Lastly, they also examined the read error rates per position. It should be noted that substitution errors are most likely associated with sequencing and not with synthesis.

3.3 SOLQC Tool

In this section we present our software tool, called SOLQC - Synthetic Oligo Library Quality Control. This quality control tool generates a customized report which consists of several statistics and plots for a given synthetic library. Detailed instructions to use the tool are given in Section 3.7.

The input to the SOLQC tool is the result of a sequencing reaction run on the library. It consists of the design variants and of all the sequenced reads. The input to the tool is provided using the following files.

1. Design file: This file consists of the design variants that were synthesized and it has to be in a csv format. The tool also supports an IUPAC description [32] of the design.

2. NGS results file: This file is in fastq format and contains the NGS results. Combining paired-end files should be done before running the tool.

3. Library configuration: additional details on the design variants such as information on the barcode indices, the length of the design variants etc.

4. More details about the input files and their format can be found in the tool’s website.

The SOLQC tool is operated in the following order.

1. Preprocessing: The reads can be filtered such that only valid reads will be processed by the tool. The selection of valid reads can be configured by the user according to the sequence barcode and its length.
2. **Matching**: Each read is matched to its corresponding variant. The matching step can be done by different strategies as follows.

- **Barcode matching**: If the library has a barcode assigned to each variant, the barcode will be used in order to match each read with a tunable tolerance in errors for the matching.

- **Edit distance**: The edit distance between an input read and, in principle, all the variants will be calculated, such that the variant with the smallest edit distance will be selected as the matched one.

- **Fast matching**: The tool supports also faster matching using several approximations of the edit distance.

Alternatively, this matching step can be done by the user in advance. In this case the matching between reads and variants is given by fourth input file (in csv format). The set of reads which are matched to the same variant form a **variant cluster**. An example file and the specific format can be found in the tool’s website.

3. **Alignment**: Every read is aligned according to its matched variant and an error vector is computed which represents the location and error types at each position of the variant (with insertions handled separately). The alignment is done using [59]. Fig. 3.1 demonstrates an example for the alignment step.

4. **Analysis**: The matched reads and their error vectors are used in order to create error characterization and data statistics for the library, as will be described in the sequel.

5. **Report generation**: The output of our tool is a report which consists of analysis results, as selected by the user, in a customizable format.
Figure 3.1: An example of 25 reads (in purple) aligned to a variant of length 27 (in yellow). For each read, the locations of the deletions, substitutions, insertions are marked in red, blue, green, respectively. This alignment output forms the basis of the analysis performed by SOLQC.
3.4 Statistical QC Analysis for Synthetic DNA Libraries

In this section we describe and discuss the statistical analysis performed and supported by the SOLQC tool. These statistics are explained on actual data from the experiment in [22] by Erlich and Zielinski. The details of this experiment are summarized in Table 3.1. These statistical results are divided into two parts; The first one addresses the composition of the synthesized library (composition statistics) and the second one addresses the errors inferred from sequencing reads (error statistics). We sampled 1,689,319 reads out of the 15,787,115 reads of the library, and analyzed only reads with length at most 4 bases shorter or longer than the design’s length, which is 152 (i.e., their base-length was between 148 and 156). The reads were filtered as a part of the preprocessing step of SOLQC as defined in Section . Those reads were matched with their closest design variants using an approximation of the edit distance which calculated the edit distance between all reads and variants based upon the first 80 bases.

Table 3.1: Experiment by Erlich and Zielinski [22]

<table>
<thead>
<tr>
<th>Data size</th>
<th>2.11 MB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Design length</td>
<td>152 bases</td>
</tr>
<tr>
<td>Number of variants</td>
<td>72000</td>
</tr>
<tr>
<td>Number of reads</td>
<td>15,787,115</td>
</tr>
<tr>
<td>Number of sampled reads</td>
<td>1,689,315</td>
</tr>
<tr>
<td>Number of filtered reads</td>
<td>1,427,781</td>
</tr>
<tr>
<td>Synthesis Technology</td>
<td>Twist Bioscience</td>
</tr>
<tr>
<td>Sequencing Technology</td>
<td>Ilumina miSeq V4</td>
</tr>
</tbody>
</table>

3.4.1 Composition statistics

1. **Symbol statistics** (Figs. 3.2 and 3.3). This plot presents, using a stacked-bar plot, the distribution of all bases in the library by their occurrence at any position both for the reads and for the design variants. Using this plot the user can compare the distribution as it reflected in the design variants and in the reads. This is demonstrated in Fig. 3.2 for the design variants and in Fig. 3.3 for the reads.

   - X-axis: The position (index) in the DNA variant or read.
• Y-axis: The number of occurrences for each base type, scaled for the sequencing depth.

• Description: In Fig. 3.2, for every position (index) in the variant, the number of occurrences of each of the four bases in all of the design variants is calculated. Similarly, in Fig. 3.3 the number in every position is calculated according to the actual reads.

Figure 3.2: Base distribution in the design variants (see 3.4.1.1).

Figure 3.3: Base distribution in the reads (see 3.4.1.1).
2. **Histogram of the cluster size per variant** (Figs. 3.4 and 3.5). The plot in Fig. 3.4 presents the histogram of the variant cluster size. That is: the number of filtered reads, per design variant.

- **X-axis:** The size of a variant cluster, starting from the size of the smallest variant cluster among all the variants in the library and up to the largest variant cluster value.
- **Y-axis:** The number of variants in the library that have a cluster of size $x$.
- **Description:** According to the matching step, the cluster size for each of the design variants is calculated and the histogram is generated by counting the number of variants with a given cluster size. Note that the sum of the $y$ values in this histogram is the number of variants in the experiment, which is 72,000 in [22].

![Histogram of the number of filtered reads per variant](image)

Figure 3.4: Histogram of the number of filtered reads per variant (see 3.4.1.2).
This plot can also have a stratified version by 5 ranges of the GC-content of the design variants, as depicted in Fig. 3.5. Using this version of the plot the user can test if there are biases related to GC-content. To define the 5 values of the GC-content presented in the figure, the tool takes the minimal and maximal values of the GC content as designed in the library and partitions the range between them to 5 different subranges of equal size (in terms of range). The GC-content is presented by percentage.

Figure 3.5: Histogram of the number of filtered reads per variant, stratified by the GC-content (see 3.4.1.2).

3. Sorted bar plot of the number of filtered reads per variant (Figs. 3.6 and 3.7). The plot in Fig. 3.6 presents a sorted bar plot for the variant cluster sizes.

- X-axis: The variant rank after sorting all variants in the library by their cluster size.
- Y-axis: The cluster size of variant $x$.
- Description: In this plot, after calculating the cluster size for each of the design variants, we sort them in a non-increasing order by the cluster size. Each variant is associated with a bar whose height corresponds to the variant cluster size. Hence, there are 72,000 bars, corresponding to the number of variants in $[22]$. 

18
Figure 3.6: Sorted bar plot of the number of filtered reads per variant (see 3.4.1.3).

We also plot, as shown in Fig. 3.7, a stratified version by 5 values of the GC-content of the variants. These 5 values of GC-content were defined by the tool as described in Fig. 3.5.

Figure 3.7: Sorted bar plot of the number of filtered reads per variant, stratified by the GC-content (see 2.2.1.3).
4. **Histogram of the length of reads** (Fig. 3.8). This plot presents the distribution of the different lengths of all the reads.

- **X-axis:** The length of the read.
- **Y-axis:** The number of filtered reads found in the library of length \( x \), presented in log-scale.
- **Description:** This plot presents a histogram of the different lengths of all reads in the library.

![Histogram of the length of reads](image)

Figure 3.8: Histogram of the length of reads (see 3.4.1.4). Note that the difference between 152 and 154 is two orders of magnitude.

### 3.4.2 Error statistics

1. **Total error rates** (Fig. 3.9). This plot presents the insertion, substitution, and deletion error rates as inferred from the reads in the library.

- **X-axis:** Each bar presents the type of error, which can be one of the following: insertions, substitutions, single-base deletions, long deletions (deletions of more than one base), and total deletions (deletions of one or more bases).
- **Y-axis:** The error rate, calculated as the ratio between the total number of errors of each type and the total number number of read bases. The plot is in log scale.
• Description: After the alignment step, an error vector is calculated for each of the reads based upon its errors with respect to the matched variant. This error vector consists of the locations of the substitutions, insertions, and deletions in the read. See Fig. 3.1 for an example. For the error rates of insertions, substitutions, and deletions, we plot the ratio between the number of occurrences of each error type (in the entire sequencing data) and the total number of read bases expected in the library (number of filtered reads × design length). For long deletions, we count each burst of at least two consecutive deletions as a single error, and then plot its ratio with the total number of read bases in the library. Lastly, the error rate of the single-base deletion is calculated in a similar way using the number of bursts of deletions of length 1.

Figure 3.9: Total error rates. (see 3.4.2.1)

2. Error rate stratified by symbol (Fig. 3.10). This plot presents by a heat map the symbol dependent, error distribution. Each square presents for each type of error, its error rate for the specific symbol. For insertions we address both the inserted symbol, and the symbol before the insertion. The $x$, $y$ entry in the heat map is calculated to be the ratio between the number of type $y$ errors of base $x$ and the expected number of base $x$ in the reads.\(^1\)

\(^1\)For example: the deletion rate in Fig. 3.1 is $24/(25 \times 27)$, which is calculated to be the ratio between the number of red squares (24) and the product of the number of rows (25) with the variant length (27).

\(^2\)The expected number of base $x$ in the reads is calculated as the sum of the products of the number of base $x$ in each of the design variants, and the number of reads matched to it.
Figure 3.10: Error rates stratified by symbol. Note that the numbers are in percents. For example, the value of 0.024 for "A" long deletion, means that 0.024 percent of the occurrences of base A in the library creates a long deletion error. (see 3.4.2.2)

3. **Cumulative distribution based upon the number of errors** (Fig. 3.11). This plot presents the percentage of reads in the library with $x$ or less errors.

- **X-axis:** Number of errors.
- **Y-axis:** Percentage of reads with at most $x$ errors.
- **Description:** For a given number of errors $x$, the tool calculates the fraction of reads with at most $x$ errors.

![Cumulative distribution](image)

Figure 3.11: Cumulative distribution based upon the number of errors (see 3.4.2.3). Note that 70% of the reads have neither sequencing nor synthesis error.

4. **Deletion length distribution** (Fig. 3.12). This plot presents the distribution of the lengths of all deletions.

- **X-axis:** Deletion length, which is the number of consecutive deleted bases.
- Y-axis: The error rate for each length of burst of deletions with exactly $x$ bases divided by the number of total bases in the library.
- Description: The tool counts the number of deletion bursts of size exactly $x$ bases, based on the alignment error vector. The error rate is then calculated as the ratio between this number and the expected number of bases in the reads.

![Deletion length distribution graph](image)

Figure 3.12: Deletion length distribution (see 3.4.2.4).

5. **Error rate per position** (Fig. 3.13). This plot presents the error rate for every error type as it is reflected in a specific position of the strand.

- X-axis: The position in the strand, from 5’ to 3’; note that the phosphoramidite synthesis direction is 3’ to 5’. It is important to emphasize that we report rates as calculated from the alignment results. These rates reflect both synthesis as well as sequencing errors. We expect substitution and insertion errors to be primarily due to sequencing. Long deletions primarily due to synthesis.
- Y-axis: The error rates per position in all reads for single-base deletions, long deletions, substitutions, and insertions, presented in log scale.
- Description: For every position between 0 (the first position, from 5’ to 3’) and 151 (the last position in [22]) and for each error type as described in Fig. 3.9, the tool calculates the error rate as the ratio between the number of errors of each type and the number of filtered reads.
6. **GC-content error analysis** (Fig. 3.14). Error rates in a form of box plot based upon the GC-content. This plot depicts the reads error rates, grouped by the GC-content of their corresponding design variants. Each point represents the error rate of one of the reads in the library, with GC-content $x$. The box extends from the lower to upper quartile error rate of the reads with GC-content $x$, and plots green line at the median and green triangle at the mean value.
Table 3.2: Synthetic DNA Libraries

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Data storage size</td>
<td>2.11 MB</td>
<td>83 KB</td>
<td>200 MB (9.50 MB)</td>
<td>3,633 bytes</td>
</tr>
<tr>
<td>Design length (bases)</td>
<td>152</td>
<td>158</td>
<td>150</td>
<td>880-1,060</td>
</tr>
<tr>
<td>Number of variants</td>
<td>72,000</td>
<td>4,991</td>
<td>607,150</td>
<td>17</td>
</tr>
<tr>
<td>Number of reads</td>
<td>15,787,115</td>
<td>3,312,235</td>
<td>62,879,612</td>
<td>6,660</td>
</tr>
<tr>
<td>Number of filtered reads</td>
<td>1,427,781</td>
<td>1,945,744</td>
<td>91,898</td>
<td>6,660</td>
</tr>
<tr>
<td>Synthesis Technology</td>
<td>Twist Bioscience</td>
<td>CustomArray</td>
<td>Twist Bioscience</td>
<td>Integrated DNA Technology (IDT)</td>
</tr>
<tr>
<td>Sequencing Technology</td>
<td>Illumina miSeq</td>
<td>Illumina miSeq</td>
<td>Illumina NextSeq</td>
<td>MinION</td>
</tr>
</tbody>
</table>

3.5 Results

In this section, we present several results from the analysis of four synthetic DNA libraries. These results are based on previous experiments for storage applications conducted by Erlich and Zielinski [22], Grass et al. [27], Organnick et al. [48], and Yazdi et al. [70]. While OLs are used for a variety of applications, we focused on data storage OLs as the library data for these is typically more accessible. We matched each read with its relevant variant using edit distance estimation as will be described below. The analyzed data sets and their details are summarized in Table 3.2. We next present how we process the data of each experiment.

3.5.1 Pre-processing and Filtering of the Libraries

- Erlich and Zielinski [22], will be referred in this paper as EZ-17: As explained in Section 3.4, we analyzed this library using a sample of 1,689,315 reads out of the reported 15,787,115 reads. In this library the design length of each variant was 152. We present example results from three different filtering schemes:
  1. Filtering only reads with length between 148 and 156 - 1,427,781 reads.
  2. Filtering only reads with length between 142 and 162 - 1,466,069 reads.
  3. Analyzing all the reads in the sample - 1,689,315 reads.

The estimated matching between each of the reads and its design variant was calculated in two steps. First, the edit distance of the first 80 bases between the read and each of the variants was calculated. Then, the read is matched with the closest variant according to this calculation.

- Grass et al. [27], will be referred in this paper as G-15: The analysis of this library is based on all of the 3,312,235 reads. The length of each variant
in this library was 158, with two primers of length 20 at the 5’ end and 21 at the 3’ end. The results presented were calculated according to the 117 bases of the data in each of the reads. The reads were filtered by their length: 1,945,744 reads with length between 112 to 122 bases were analyzed by the tool. The estimated matching of the reads to their corresponding design variants in G-15 was performed as in EZ-17.

- Organick et al. [48], will be referred in this paper as O-17: The analysis of this library is based on a sample of 101,243 out of the 62,879,612 reads of one file of the library. The design length of each variant in this library was 150. Similarly to G-15 [27], there were two primers of length 20 at each end. Hence, the reads were filtered by their length: we omitted the primers from each read, and analyzed 91,898 reads with length between 105 and 115 bases. The results are presented for the information bases (the primers were trimmed). The estimated matching of the reads in O-17 was performed as in EZ-17.

- Yazdi et al. [70], will be referred in this paper as Y-16: The results presented are based on all the 6,660 reads in the library. This library consists of 17 variants - 15 of length 1,000, one of length 1060 and one of length 880. The estimated matching of the reads to their corresponding design variants, was done in a similar way that used for EZ-17. However, since the number of variants was significantly smaller, we were able to calculate the edit distance for the entire strand between each read and all of the variants. Then, the read was matched with the closest variant. In this experiment, the design variants were longer and the reads were sequenced by the MinION sequencing technology. Hence, this data is likely have different error characteristics than those observed for the other three.

3.5.2 Analysis of Synthetic DNA Datasets

1. Total error rates. The results show significant differences between the four experiments. The three experiments of EZ-17 [22], G-15 [27], and O-17 [48] show higher rates for deletions and substitutions than insertions. EZ-17 and O-17 have the lowest error rates overall. In Y-16 [70], we observe higher rates for insertions rather than deletions and substitutions. Moreover, the error rates in Y-16 [70] are higher by two orders of magnitude than the other.
three. These results are presented in Fig. 3.15.

2. Cumulative distributions based upon the number of errors. As mentioned above the data of Y-16 [70] is much more erroneous. Indeed, we can see that none of its reads had less than 100 errors. In EZ-17 [22] and O-17 [48], 70% and 60% of the reads were synthesized and sequenced without any error respectively, while only 30% of the reads in G-15 [27] show no errors at all. These results are presented in Fig. 3.16.

3. Error rates, stratified by symbol. Base G showed slightly less errors compared to the other bases in EZ-17 [22]. Similarly, base A and base G showed slightly lower error rates compared to the other bases in Y-16 [70]. However, in G-15 [27], and in O-17 [48], base C was the least erroneous base. These results are presented in Fig. 3.17.

4. Histograms of the length of the reads, using different filtering schemes in EZ-17 [22]. We can see that in each of the filtering schemes we used, the length of the majority of the reads was 152 (the designed length) or shorter. These results correspond to our findings that deletions were the most dominant errors in the library. These results are presented in Fig. 3.18.

5. Histograms of the cluster size per variant. Fig. 3.19 shows the distributions of the cluster size per variant for the experiments in EZ-17 [22] and G-15 [27]. While the shape of the distribution of EZ-17 [22] has the form of a normal distribution, there is no similar trend in G-15 [27]. However, it is possible to notice that the number of variants decreases with the size of the cluster size. Furthermore, in G-15 [27] we also observed very large clusters of size ranging between 2,000 and 8,000, which were omitted from the figure for its clarity.

6. Error rates by GC-content. The results (presented in Fig. 3.20) show that the median and the mean values of the read error rates increase with its designed GC-content in G-15 [27]. In Y-16 [70] we can surprisingly see error rates which are greater than 1. Such high error rates are encountered when there is a large number of insertions together with deletions and substitutions in the read such that the number of errors is strictly larger than the design length. These results corresponding to our findings that insertions were the most dominant errors in this library.
7. **Error rate per position.** In all four experiments analyzed, the error rates in the 3’ end are greater than the error rates in the 5’ end. Note that in Y-16 [70] there were different design lengths: 880, 1,000 and 1,060. For uniformity, we present only results of reads which correspond to variants of length 1,000. These results are presented in Fig. 3.21.

### 3.6 Use-Case Examples

In this section we present several use-case examples for SOLQC.

1. **Design-quality evaluation.** Different libraries can have different robustness levels. For example: a design of one library can have many homopolymers, while another can limit the presence of homopolymers. SOLQC outputs statistical reports describing the error behavior of a given library/design. The user can create several small test experiments with different designs and properties. Then, the user can use SOLQC to evaluate the effect of different designs on the error behavior. This analysis can then be considered as part of the final design of the library.

2. **Binning of synthetic DNA-libraries.** The result of a sequencing reaction on a given library does not include the matching of each read to its design variant. SOLQC provides several methods to bin the reads according to their corresponding design variants. The matching/clustering methods can be performed on libraries with or without the barcode. In addition, users can get coverage depth statistics from SOLQC as well as quality related statistics, which can be different for different variants or set of variants. Lastly, in applications like data storage, the set of reads that is binned to any given variant can be used in order to decode the stored variant.

3. **Comparison of different synthesis and sequencing technologies.** SOLQC provides its users composition and errors statistics. Accordingly, the user can synthesize libraries using several synthesis technologies and their process parameters. Then, the user can compare the quality of the results of each technology and/or of each parameter configuration. In order to optimize the process parameters, the experiments can be conducted with the same design while using different parameter configuration. Thus, it is possible to determine how to choose the best configuration.
4. **Design of error-correcting codes and coding techniques for DNA-storage.**
   In data storage applications, SOLQC can be used as a characterization tool of the DNA channel. The user can characterize the DNA channel using data from previous experiments of various technologies and design parameters. Then, using this information, the user can design appropriate error-correcting codes and coding techniques to improve the error rates.

5. **Standardization and reproducibility.** SOLQC enables determining whether a library is behaving as previous libraries from the same vendor with similar preparation characteristics. This enables comparison between the same library preparation protocol performed in different labs, or in the same lab at different times or by different lab members.

### 3.7 Installation

Detailed instruction of SOLQC installation, can be found in SOLQC installation guidelines in our website: [SOLQC installation guide](#).

#### 3.7.1 Installing Docker

The first step of SOLQC’s installation is Docker installation. Docker is a computer program that performs operating-system-level virtualisation. Docker enables running of a virtual machine from the command line (terminal). Using this virtual machine, a user can run SOLQC with just a single command. Docker installation instruction can be found here: [Docker getting started](#).

A user should follow these three steps for installing Docker before running SOLQC tool:

1. Sign-Up for docker.
2. Download and install docker.
3.7.2 Installing SOLQC

Once Docker is running, the following line should be type from the terminal to download and start SOLQC:

```
docker run -p 5000:5000 solqc/tool
```

The first time a user run this line, it will download SOLQC, so this might take a couple of minutes. Once it finish the download it will start the software automatically. Lastly, to access SOLQC, a user should open a web browser and go to the following address: [http://localhost:5000/home](http://localhost:5000/home)

If all went well, the following screen should be seen:

![SOLQC Interface](image-url)

If you use this tool please cite this paper.
Figure 3.15: Total error rates in the four datasets. Note that total error rates are also due to sequencing errors. While EZ-17 [22], G-15 [27], and O-17 [48] were sequenced on Illumina sequencing machines, the fourth dataset, Y-16 [70], used a much noisier sequencing platform, which explains the differences in the error rates.
Figure 3.16: Cumulative distributions based upon the number of errors. Note differences in the range of Y-axes in the four figure. Also note that the X-axes are truncated (see 3.4.2.3).
Figure 3.17: Error rates, stratified by symbol in the four datasets (see 3.4.2.2). Note that the colorbars are different in each plot.
Figure 3.18: Histograms of the length of the reads in EZ-17 [22], using different filtering schemes (1), (2), and (3), respectively (see 3.4.1.4) .
Figure 3.19: Histograms of the cluster size per variant in two of the four datasets (see 3.4.1.2).
Figure 3.20: Error rates by the GC-content in the four datasets. Note the differences in the range of the Y-axis in the four figures (see 3.4.2.6).
Figure 3.21: Error rates per position in the four datasets. X-axis represents position counted from the 5’ end of the designed variant, note the differences in the range of the X-axis and Y-axis in the four figures (see 3.4.2.5).
Chapter 4

The Error Probability of Maximum Likelihood Decoding over Two Deletion/Insertion Channels

Motivated by the reconstruction problem and its application for DNA-storage systems, in this chapter we calculate the error probability and the failure probability of the maximum-likelihood (ML) decoder over two deletion/insertion channels. For two deletion channels, we also study the failure probability of the ML decoder, when the transmitted word belongs to the Varshamov Tenegolts (VT) \[67\] code or the shifted VT code \[56\]. We also verify our results by simulations.

4.1 Definitions and Preliminaries

We denote by $\Sigma_q = \{0, \ldots, q - 1\}$ the alphabet of size $q$ and $\Sigma^*_q \triangleq \bigcup_{\ell=0}^{\infty} \Sigma^\ell_q$, $\Sigma^{\leq n}_q \triangleq \bigcup_{\ell=0}^{n} \Sigma^\ell_q$, $\Sigma^{\geq n}_q \triangleq \bigcup_{\ell=n}^{\infty} \Sigma^\ell_q$. The length of $x \in \Sigma^n$ is denoted by $|x| = n$. The Levenshtein distance between two words $x, y \in \Sigma^*_q$, denoted by $d_L(x, y)$, is the minimum number of insertions and deletions required to transform $x$ into $y$, and $d_H(x, y)$ denotes the Hamming distance between $x$ and $y$, when $|x| = |y|$. A word $x \in \Sigma^*_q$ will be referred to as an alternating sequence if it cyclically repeats all symbols in $\Sigma_q$ in the same order. For example, for $\Sigma_2 = \{0, 1\}$, the two alternating sequences are 010101\ldots and 101010\ldots, and in general there are $q!$
alternating sequences. For $n \geq 1$, the set $\{1, \ldots, n\}$ is abbreviated by $[n]$.

For a word $x \in \Sigma_\neq^n$ and a set of indices $I \subseteq [\|x\|]$, the word $x_I$ is the projection of $x$ on the indices of $I$ which is the subsequence of $x$ received by the symbols in the entries of $I$. A word $x \in \Sigma^*$ is called a supersequence of $y \in \Sigma^*$, if $y$ can be obtained by deleting symbols from $x$, that is, there exists a set of indices $I \subseteq [\|x\|]$ such that $y = x_I$. In this case, it is also said that $y$ is a subsequence of $x$. Furthermore, $x$ is called a common supersequence (subsequence) of some words $y_1, \ldots, y_t$ if $x$ is a supersequence (subsequence) of each one of these $t$ words. The set of all common supersequences of $y_1, \ldots, y_t \in \Sigma_\neq^n$ is denoted by $\text{SCS}(y_1, \ldots, y_t)$ and $\text{SCS}(y_1, \ldots, y_t)$ is the length of the shortest common supersequence (SCS) of $y_1, \ldots, y_t$, that is, $\text{SCS}(y_1, \ldots, y_t) = \min_{x \in \text{SCS}(y_1, \ldots, y_t)} \|x\|$. Similarly, $\text{LCS}(y_1, \ldots, y_t)$ is the set of all subsequences of $y_1, \ldots, y_t$ and we let $\text{LCS}(y_1, \ldots, y_t)$ be the length of the longest common subsequence (LCS) of $y_1, \ldots, y_t$, that is, $\text{LCS}(y_1, \ldots, y_t) = \max_{x \in \text{LCS}(y_1, \ldots, y_t)} \|x\|$.

We consider a channel $S$ that is characterized by a conditional probability $\Pr_S$, which is defined by

$$\Pr_S \{ y \text{ rec.} | x \text{ trans.} \},$$

for every pair $(x, y) \in (\Sigma_\neq^n)^2$. Note that it is not assumed that the lengths of the input and output words are the same as we consider also deletions and insertions of symbols, which is the main topic of this work. As an example, it is well known that if $S$ is the binary symmetric channel ($\text{BSC}$) with crossover probability $0 \leq p \leq 1/2$, denoted by $\text{BSC}(p)$, it holds that

$$\Pr_{\text{BSC}(p)} \{ y \text{ rec.} | x \text{ trans.} \} = p^{d_{\text{hit}}(y,x)}(1-p)^{n-d_{\text{hit}}(y,x)},$$

for all $(x, y) \in (\Sigma_\neq^n)^2$, and otherwise (the lengths of $x$ and $y$ is not the same) this probability equals 0. Similarly, for the $Z$-channel, denoted by $Z(p)$, it is assumed that only a 0 can change to a 1 with probability $p$ and so

$$\Pr_{Z(p)} \{ y \text{ rec.} | x \text{ trans.} \} = p^{d_{\text{hit}}(y,x)}(1-p)^{n-d_{\text{hit}}(y,x)},$$

for all $(x, y) \in (\Sigma_\neq^n)^2$ such that $x \leq y$, and otherwise this probability equals 0.

In the deletion channel with deletion probability $p$, denoted by $\text{Del}(p)$, every symbol of the word $x$ is deleted with probability $p$. Similarly, in the insertion channel with insertion probability $p$, denoted by $\text{Ins}(p)$, a symbol is inserted in each of the possible $|x| + 1$ positions of the word $x$ with probability $p$, while the
probability to insert each of the symbols in $\Sigma_q$ is the same and equals $\frac{P}{q}$.

A decoder for a code $C$ with respect to the channel $S$ is a function $D : \Sigma_q^* \to C$. Its average decoding failure probability is defined by $P_{\text{fail}}(S, C, D) = \frac{\sum_{c \in C} P_{\text{fail}}(c)}{|C|}$, where

$$P_{\text{fail}}(c) = \sum_{y : D(y) \neq c} \Pr_S\{y \text{ rec.} | c \text{ trans.}\}.$$  

We will also be interested in the average decoding error probability which is the average normalized distance between the transmitted word and the decoder’s output. The distance will depend upon the channel of interest. For example, for the BSC we will consider the Hamming distance, while for the deletion and insertion channels, the Levenshtein distance will be of interest. Hence, for a channel $S$, distance function $d$, and a decoder $D$, we let $P_{\text{err}}(S, C, D, d) = \frac{\sum_{c \in C} P_{\text{err}}(c,d)}{|C|}$, where

$$P_{\text{err}}(c,d) = \sum_{y : D(y) \neq c} \frac{d(D(y), c)}{|c|} \cdot \Pr_S\{y \text{ rec.} | c \text{ trans.}\}.$$  

The maximum-likelihood (ML) decoder for $C$ with respect to $S$, denoted by $D_{\text{ML}}$, outputs a codeword $c \in C$ that maximizes the probability $\Pr_S\{y \text{ rec.} | c \text{ trans.}\}$. That is, for $y \in \Sigma_q^*$,

$$D_{\text{ML}}(y) = \arg\max_{c \in C} \{\Pr_S\{y \text{ rec.} | c \text{ trans.}\}\}.$$  

It is well known that for the BSC, the ML decoder simply chooses the closest codeword with respect to the Hamming distance. The channel capacity is referred to as the maximum information rate that can be reliably transmitted over the channel $S$ and is denoted by $\operatorname{Cap}(S)$. For example, $\operatorname{Cap}(\text{BSC}(p)) = 1 - H(p)$, where $H(p) = -p \log_2(p) - (1 - p) \log_2(1 - p)$ is the binary entropy function.

The conventional setup of channel transmission is extended to the case of more than a single instance of the channel. Assume a word $x$ is transmitted over some $t$ identical channels of $S$ and the decoder receives all channel outputs $y_1, \ldots, y_t$. This setup is characterized by the conditional probability

$$\Pr_{(S,t)}\{y_1, \ldots, y_t \text{ rec.} | x \text{ trans.}\} = \prod_{i=1}^t \Pr_S\{y_i \text{ rec.} | x \text{ trans.}\}.$$  

The definitions of a decoder, the ML decoder and the error probabilities are ex-
tended similarly. The input to the ML decoder is the words \( y_1, \ldots, y_t \) and the output is the codeword \( c \) which maximizes the probability \( \Pr_S(y_1, \ldots, y_t \text{ rec.} | x \text{ trans.}) \). The average decoding failure probability, average decoding error probability is generalized in the same way and is denoted by \( P_{\text{fail}}(S, t, C, D) \), \( P_{\text{err}}(S, t, C, D, d) \), respectively. The capacity of this channel is denoted by \( \text{Cap}(S, t) \), so \( \text{Cap}(S, 1) = \text{Cap}(S) \).

The case of the BSC was studied by Mitzenmacher in [46], where he showed that

\[
\text{Cap}(\text{BSC}(p), t) = 1 + \sum_{i=0}^{t} \binom{t}{i} \left( p^i (1-p)^{t-i} \log \frac{p^i (1-p)^{t-i}}{p^i (1-p)^{t-i} + p^{t-i} (1-p)^i} \right).
\]

On the other hand, the \( Z \) channel is significantly easier to solve and it is possible to verify that \( \text{Cap}(\text{Z}(p), t) = \text{Cap}(\text{Z}(p^t)) \). It is also possible to calculate the average decoding error and failure probabilities for the BSC and \( Z \) channels. For example, when \( C = \Sigma_2^n \), one can verify that

\[
P_{\text{err}}(\text{BSC}(p), t, \Sigma_2^n, D_{\text{ML}}, d_H) = p^t,
\]

and if \( t \) is odd then

\[
P_{\text{err}}(\text{BSC}(p), t, \Sigma_2^n, D_{\text{ML}}, d_H) = \sum_{i=0}^{t-1} \binom{t}{i} p^{t-i} (1-p)^i.
\]

Similarly, \( P_{\text{fail}}(\text{BSC}(p), t, \Sigma_2^n, D_{\text{ML}}) = 1 - (1 - p^t)^n \) and \( P_{\text{fail}}(\text{BSC}(p), t, \Sigma_2^n, D_{\text{ML}}) = 1 - (1 - \sum_{i=0}^{t-1} \binom{t}{i} p^{t-i} (1-p)^i)^n \). However, calculating these probabilities for the deletion and insertion channels is a far more challenging task. The goal of this chapter is to study in depth the special case of \( t = 2 \) and estimate the average error and failure probabilities, when the code is the entire space, the Varshamov Tenengolts (VT) code [67], and the shifted VT (SVT) code [56].

This model is closely connected to several related problems. In the reconstruction problem studied by Levenshtein [39, 40], it was assumed that the word is transmitted over several noisy channels and the goal of the decoder is to decode the transmitted word in the worst case, assuming that all channels’ outputs are different from each other. Several extensions of these problems have been studied; see e.g. [26, 41, 52, 55, 58, 59], however in all of them the goal is to find the
number of channels that guarantees unique decoding in the worst case. The most relevant case of the reconstruction problem to our work is the one studied in [56], where it was shown how the shifted VT codes can be used for the two single-deletion channels case. In a parallel work [34] the dual problem is studied where the number of channels is given and then the goal is to find the best code which guarantees successful decoding in the worst case. Hence, the problem studied in this paper can be regarded as the probabilistic variant of the dual problem of the reconstruction problem. Yet another highly related problem is the one of the trace reconstruction problem [5, 18, 19, 29, 30, 47, 51]. The most relevant works to our study are the recent ones [60, 61], where decoding algorithms for maximum likelihood are presented for a fixed number of channels. A comprehensive summary of the previous work regarding the trace reconstruction problem, and its applications to DNA-storage systems can be found in Section 5.2.

4.2 The Deletion and Insertion Channels

In this section we establish several basic results for the deletion channel with multiple instances. We start with several useful definitions. For two words $x, y \in \Sigma_q^*$, the number of times that $y$ can be received as a subsequence of $x$ is called the embedding number of $y$ in $x$ and is defined by

$$\text{Emb}(x; y) = |\{I \subseteq [|x|] \mid x_I = y\}|.$$

Note that if $y$ is not a subsequence of $x$ then $\text{Emb}(x; y) = 0$. The embedding number has been studied in several previous works; see e.g. [3, 21] and in [60] it was referred to as the binomial coefficient. In particular, this value can be computed with quadratic complexity [21].

While the calculation of the conditional probability $\Pr_S\{y \text{ rec.} \mid x \text{ trans.}\}$ is a rather simple task for many of the known channels, it is not straightforward for channels which introduce insertions and deletions. The following basic claim is well known and was also stated in [60], however it is presented here for the completeness of the results in the paper and since it will be used in our derivations to follow.
Claim 4.1. For all \((x, y) \in (\Sigma_q^*)^2\), it holds that
\[
\Pr_{\text{Del}(p)}\{y \text{ rec.} | x \text{ trans.}\} = p^{|x| - |y|} \cdot (1 - p)^{|y|} \cdot \Emb(x; y),
\]
\[
\Pr_{\text{Ins}(p)}\{y \text{ rec.} | x \text{ trans.}\} = \left(\frac{p}{q}\right)^{|y| - |x|} \cdot (1 - p)^{|x| + 1 - (|y| - |x|))} \cdot \Emb(y; x).
\]

According to Claim 4.1, it is possible to explicitly characterize the ML decoder for the deletion and insertion channels as described also in [60].

Claim 4.2. Assume \(c \in C \subseteq (\Sigma_q)^n\) is the transmitted word and \(y \in (\Sigma_q)^{\leq n}\) is the output of the deletion channel \(\text{Del}(p)\), then
\[
\mathcal{D}_{\text{ML}}(y) = \arg \max_{c \in C} \{ \Emb(c; y) \}.
\]

Similarly, for the insertion channel \(\text{Ins}(p)\), for \(y \in (\Sigma_q)^{\geq n}\),
\[
\mathcal{D}_{\text{ML}}(y) = \arg \max_{c \in C} \{ \Emb(y; c) \}.
\]

In case there is more than a single instance of the deletion channel, the following claim follows.

Claim 4.3. Assume \(c \in C \subseteq (\Sigma_q)^n\) is the transmitted word and \(y_1, \ldots, y_t \in (\Sigma_q)^{\leq n}\) are the output words from \(\text{Del}(p)\), then
\[
\mathcal{D}_{\text{ML}}(y_1, \ldots, y_t) = \arg \max_{c \in C \cap \mathcal{CS}(y_1, \ldots, y_t)} \left\{ \prod_{i=1}^{t} \Emb(c; y_i) \right\},
\]
and for the insertion channel \(\text{Ins}(p)\), for \(y_1, \ldots, y_t \in (\Sigma_q)^{\geq n}\),
\[
\mathcal{D}_{\text{ML}}(y_1, \ldots, y_t) = \arg \max_{c \in C \cap \mathcal{CS}(y_1, \ldots, y_t)} \left\{ \prod_{i=1}^{t} \Emb(y_i; c) \right\}.
\]

Proof Every candidate to be considered in the ML decoder is a common super-sequence of \(y_1, \ldots, y_t\). Hence, \(\mathcal{D}_{\text{ML}}(y) = c\), where \(c \in C \cap \mathcal{CS}(y_1, \ldots, y_t)\) maximizes
\[
\Pr_{\text{Del}(p)}\{y_1, \ldots, y_t \text{ rec.} | c \text{ trans.}\} = \prod_{i=1}^{t} p^{|c| - |y_i|} \cdot (1 - p)^{|y_i|} \cdot \Emb(c; y_i).
\]
Since \(\prod_{i=1}^{t} p^{|c| - |y_i|} \cdot (1 - p)^{|y_i|}\) is the same for all candidates \(c\), the statement holds. A similar proof holds for the insertion channel.
Note that since there is more than a single channel, when the goal is to minimize the average decoding error probability, the ML decoder does not necessarily have to output a codeword but any word that minimizes the average decoding error probability. Thus, for the rest of the paper, when discussing the average decoding error probability it is assumed that the ML decoder can output any word and not necessarily a codeword from $C$. Thus, we get the following claim.

**Claim 4.4.** Assume $c \in C \subseteq (\Sigma_q)^n$ is the transmitted word and $y_1, \ldots, y_t \in (\Sigma_q)^{\leq n}$ are the output words from $\text{Del}(p)$, then

$$D_{\text{ML}}(y_1, \ldots, y_t) = \arg\max_{x \in \text{SCS}(y_1, \ldots, y_t)} \left\{ p^{x \cdot t} \prod_{i=1}^{t} \text{Emb}(x; y_i) \right\},$$

and for the insertion channel $\text{Ins}(p)$, for $y_1, \ldots, y_t \in (\Sigma_q)^{\geq n}$,

$$D_{\text{ML}}(y_1, \ldots, y_t) = \arg\max_{x \in \text{LCS}(y_1, \ldots, y_t)} \left\{ p^{x \cdot t} \prod_{i=1}^{t} \text{Emb}(y_i; x) \right\}.$$

Assume $C$ is $\Sigma_q^n$. The average decoding failure probability of the ML decoder over the deletion channel $\text{Del}(p)$ with $t$ instances is denoted by $P_{\text{fail}}(\text{Del}(p), t, \Sigma_q^n, D_{\text{ML}})$ and shortly $P_{\text{fail}}(q, p, t)$. Similarly, the average decoding error probability is $P_{\text{err}}(\text{Del}(p), t, \Sigma_q^n, D_{\text{ML}}, d_L)$ and shortly $P_{\text{err}}(q, p, t)$. If $t = 2$ it will be removed from the notations.

Our main goal in the rest of the chapter is to calculate close approximations for $P_{\text{fail}}(q, p, t)$ and $P_{\text{err}}(q, p, t)$ when $t = 2$. Note that a lower bound on these probabilities is $p^t$ since if the same symbol is deleted in all of the channels, then it is not possible to recover its value and thus it will be deleted also in the output of the ML decoder. This was already observed in [60] and in their simulation results. In the next section, we will analyze these probabilities for the special case of $t = 2$, when the code is $\Sigma_q^n$, the VT code [67], and the SVT code [56].

Complexity wise, it is well known that the time complexity to calculate the SCS length and the embedding numbers of two sequences are both quadratic with their length. However, the number of SCSs can grow exponentially [21][31]. Thus, given a list of SCSs of size $L$, the complexity of the ML decoder for $t = 2$ will be $O(Ln^2)$. The main idea behind these algorithms uses dynamic programming in order to calculate the SCS length and the embedding numbers for all prefixes of the given words. However, when calculating for example the SCS for $y_1$ and $y_2$ it is already known that $\text{SCS}(y_1, y_2) \leq n$. Hence, it is not hard to observe that (see e.g. [2]) many paths corresponding to prefixes which their length difference is
greater than \( d_1 + d_2 \) can be eliminated, when \( d_1, d_2 \) is the number of deletions in \( y_1, y_2 \), respectively. In particular, when \( d_1 \) and \( d_2 \) are fixed, then the time complexity is linear. In our simulations we used this improvement when implementing the ML decoder. Other improvements and algorithms of the ML decoder are discussed in [60, 61].

### 4.3 Two Deletion Channels

In this section we consider only the case of two deletion channels and prove in Theorem 4.8 an approximation for the average decoding error probability in the form of

\[
P_{err}(q, p) \approx \frac{3q - 1}{q - 1}p^2 + O(p^3).
\]

As mentioned in Section 4.2 a lower bound on the value of \( P_{err}(q, p, t) \) is \( p^t \). This lower bound is indeed not tight since if symbols from the same run are deleted then the outputs of the two channels of this run are the same and it is impossible to detect that this run experienced a deletion in both of its copies. The probability of deletions due to runs is denoted by \( P_{run}(q, p) \) and the next lemma approximates this probability.

**Lemma 4.5.** For the deletion channel \( Del(p) \), it holds that

\[
P_{run}(q, p) \approx \frac{q + 1}{q - 1}p^2.
\]
Proof Given a run of length \( r \), the probability that both of its copies have experienced a deletion is roughly \((rp)^2\). Furthermore, the occurrence probability of a run of length exactly \( r \) is \((\frac{1}{q})^{r-1} \cdot \frac{q-1}{q}\). Thus, for \( n \) large enough, the error probability is approximated by

\[
\sum_{r=1}^{\infty} (rp)^2 \left( \frac{1}{q} \right)^{r-1} \cdot \frac{q-1}{q} = p^2 \cdot \frac{q-1}{q} \sum_{r=1}^{\infty} r^2 \left( \frac{1}{q} \right)^{r-1} = p^2 \cdot \frac{q-1}{q} \cdot (1 - \frac{1}{q}) = p^2 \cdot \frac{(q+1)}{(q-1)^2}.
\]

The expected length of a run is given by

\[
\sum_{r=1}^{\infty} r \left( \frac{1}{q} \right)^{r-1} \cdot \frac{q-1}{q} = \frac{q-1}{q} \sum_{r=1}^{\infty} r \left( \frac{1}{q} \right)^{r-1} = \frac{q}{q-1}.
\]

Hence, the expected number of runs in a vector of length \( n \) is \( n \cdot \frac{q-1}{q} \) and thus the approximated number of deletions due to runs is

\[
n \cdot \frac{q-1}{q} \cdot p^2 \cdot \frac{q(q+1)}{(q-1)^2} = np^2 \cdot \frac{q+1}{q-1},
\]

which verifies the statement in the lemma.

However, runs are not the only source of errors in the output of the ML decoder. For example, assume the \( i \)-th and the \( (i+1) \)-th symbols are deleted from the two channels. If the transmitted word \( x \) is of the form \( x = (x_1, \ldots, x_i-1, 0, 1, x_{i+2}, \ldots, x_n) \), then the two channels’ outputs are \( y_1 = (x_1, \ldots, x_i-1, 0, x_{i+2}, \ldots, x_n) \) and \( y_2 = (x_1, \ldots, x_i-1, 1, x_{i+2}, \ldots, x_n) \). However, these two outputs could also be received upon deletions exactly in the same positions if the transmitted word is \( x' = (x_1, \ldots, x_i-1, 1, 0, x_{i+2}, \ldots, x_n) \). Hence, the ML decoder can output the correct word only in one of these two cases. Longer alternating sequences cause the same problem as well and the occurrence probability of this event, denoted by \( P_{alt}(q, p) \), will be estimated in the next lemma.

Lemma 4.6. For the deletion channel \( \text{Del}(p) \), it holds that

\[
P_{alt}(q, p) \approx 2p^2.
\]

46
Proof Assume there is a deletion in the first channel in the \(i\)-th position and the closest deletion in the second channel is \(j > 0\) positions apart, i.e., either in position \(i - j\) or \(i + j\). For simplicity assume it is in the \((i + j)\)-th position and \(x_{[i:i+j]}\) is an alternating sequence \(ABAB \cdots\). Then, the same outputs from the two channels could be received if the transmitted word is the same as \(x\) but with changing the order of the alternating sequence, that is, the symbols of the word in the positions of \([i : j]\) are \(BABA \cdots\). Therefore, the occurrence probability of this event can be approximated by

\[
2p^2 \cdot \sum_{j=1}^{\infty} \frac{q-1}{q} \cdot \frac{1}{q^{j-1}} = 2p^2,
\]

where \(\frac{q-1}{q} \cdot \frac{1}{q^{j-1}}\) is the probability that \(x_{[i:i+j]}\) is any alternating sequence and the multiplication by 2 takes into account the cases of deletion in either position \(i - j\) or \(i + j\).

At this point one may ask whether these two error events are the only dominant ones and indeed this question is answered in the affirmative, as stated in the following lemma.

**Lemma 4.7.** If there is a deletion in the first, second channel in position \(i, i + j\), where \(j > 0\), respectively, and the sequence \(x_{[i:i+j]}\) is neither a run nor an alternating sequence, then these deletions are corrected successfully by the ML decoder.

**Proof** The output of the first channel is \((x_{i+1}, x_{i+2}, \ldots, x_{i+j})\) and the one of the second channel is \((x_i, x_{i+1}, \ldots, x_{i+j-1})\). Clearly, it holds that \((x_i, x_{i+1}, x_{i+2}, \ldots, x_{i+j})\) is a candidate for the ML decoder. If the vector \((x'_i, x'_{i+1}, x'_{i+2}, \ldots, x'_{i+j})\) is another candidate for the ML decoder, since the deletions occured in position \(i, i + j\), it should hold that

\[
(x'_i, x'_{i+1}, x'_{i+2}, \ldots, x'_{i+j-1}) = (x_{i+1}, x_{i+2}, \ldots, x_{i+j})
\]

and

\[
(x'_{i+1}, x'_{i+2}, \ldots, x'_{i+j}) = (x_i, x_{i+1}, \ldots, x_{i+j-1}).
\]

Therefore we get that \(x_i = x_{i+2} = x_{i+4} = \cdots\) and \(x_{i+1} = x_{i+3} = x_{i+5} = \cdots\), i.e., \(x_{[i:i+j]}\) is either the alternating sequence or a run, which results with a contradiction.
We are now ready to show the following theorem on the Levenshtein error rate for the case of two channels.

**Theorem 4.8.** The Levenshtein error rate for two deletion channels is approximated by

\[
P_{\text{err}}(q, p) \approx P_{\text{run}}(q, p) + P_{\text{alt}}(q, p) + O(p^3) = \frac{3q - 1}{q - 1} p^2 + O(p^3).
\]

**Proof** The proof follows from the above few lemmas. Note that each run error translates to an increase of the Levenshtein distance by one. On each occurrence of the alternating event the decoder chooses the correct subsequence with probability 0.5 and every error increases the Levenshtein distance by two since it translates to one insertion and one deletion. Lastly, the \(O(p^3)\) expression compensates for all other less dominant error events which introduce more than two deletions that are close to each other at least in one of the channels.

Using these observations, we are also able to approximate the average decoding failure probability.

**Theorem 4.9.** The average decoding failure probability is

\[
P_{\text{fail}}(q, p) \approx e^{-\frac{3q - 1}{q - 1} p^2 n}.
\]

**Proof** This is the probability that there was neither a deletion error because of the runs nor errors because of the alternating sequences. Hence, this probability becomes

\[
(1 - P_{\text{run}}(q, p))^n \cdot (1 - P_{\text{alt}}(q, p))^n \\
\approx (1 - (P_{\text{run}}(q, p) + P_{\text{alt}}(q, p)))^n = \left(1 - \frac{3q - 1}{q - 1} p^2\right)^n \\
\approx e^{-\frac{3q - 1}{q - 1} p^2 n}.
\]

So far we have discussed only the case in which the code \(C\) is the entire space. However, the most popular deletion-correcting code is the VT code \([67]\). Recently, SVT, an extension of the VT code, has been proposed in \([56]\) for the correction of burst deletions. The goal of the SVT code is to correct a deletion error while its position is known up to some roughly \(\log(n)\) consecutive locations. However, this construction has been recently used in \([12]\) to build a code that is
specifically targeted for the reconstruction of a word that is transmitted through two single-deletion channels. Due to the relevance of correcting deletion and alternating errors, the decoding failure probabilities of these two codes are investigated in this work. We abbreviate the notation of $P_{\text{fail}}(\text{Del}(p), 2, VT_n, D_{ML}), P_{\text{fail}}(\text{Del}(p), 2, SVT_n, D_{ML})$ by $P_{\text{fail}}(VT_n, q, p), P_{\text{fail}}(SVT_n, q, p)$, respectively. The following theorem summarizes these results.

**Theorem 4.10.** The average decoding failure probabilities for the VT and SVT codes are given by

$$P_{\text{fail}}(VT_n, q, p) \approx (1 - P_{\text{run}}(q, p))^n \cdot (1 - P_{\text{alt}}(q, p))^n + (1 - P_{\text{run}}(q, p))^n \cdot nP_{\text{alt}}(q, p) (1 - P_{\text{alt}}(q, p))^{n-1},$$

$$+ nP_{\text{run}}(q, p) (1 - P_{\text{run}}(q, p))^{n-1} \cdot (1 - P_{\text{alt}}(q, p))^n.$$

$$P_{\text{fail}}(SVT_n, q, p) \approx (1 - P_{\text{run}}(q, p))^n \cdot (1 - P_{\text{alt}}(q, p))^n + (1 - P_{\text{run}}(q, p))^n \cdot nP_{\text{alt}}(q, p) (1 - P_{\text{alt}}(q, p))^{n-1}.$$

**Proof** The proof follows from the observation that the VT code is capable of decoding either a single run error or a single alternating error. On the other hand, the shifted VT code is capable of only correcting a single alternating error; see [12] for more details.

We verified the theoretical results presented in this section by the following simulations. These simulations were tested over words of length $n = 450$ which were used to create two noisy copies given a fixed deletion probability $p \in [0.005, 0.05]$. Then, the two copies were decoded by the ML decoder as described in Claim 4.4. Finally, we calculated the Levenshtein error rate of the decoded word as well as the average decoding failure probability (referred to here as *failure rate*). Fig. 4.1 plots the results of the Levenshtein error rate which confirms the probability expression $P_{\text{err}}(q, p)$ from Theorem 4.8. Similarly, in Fig. 4.2 we present separately the error probability $P_{\text{alt}}(q, p), P_{\text{run}}(q, p)$, along with the corresponding value as calculated in Lemma 4.5, 4.6 respectively. Lastly, in Fig. 4.3, we simulated the ML decoder for the VT codes and the SVT codes and calculated the failure rate $P_{\text{fail}}(q, p)$. The implementation of the VT code was taken from [64], and we modified this implementation for SVT codes.

49
Figure 4.1: Levenshtein error rate by the deletion probability $p$. These simulations verify Theorem 4.8.

Figure 4.2: Levenshtein error rate by the deletion probability $p$. These simulations verify Lemma 4.5 and Lemma 4.6.
4.4 Two Insertion Channels

This section continues the two-channel study but for the insertion case. In a similar manner to the deletion case, also here the dominant errors result from increasing the length of a run and error that results from the occurrence of an alternating sequence. We denote by $P_{\text{ins}}^{\text{err}}(q, p)$ the Levenshtein error rate of the ML decoder upon two instances of the insertion channel $\text{Ins}(p)$. Similarly to the deletion case, $P_{\text{run}}^{\text{ins}}(q, p), P_{\text{alt}}^{\text{ins}}(q, p)$ is the insertion probability due to runs, occurrence probability due to alternate sequences, respectively. Theorem 4.14 summarizes the results of this section. The proof of this theorem repeats the same ideas as the ones for the deletion case. However, we list here the relevant lemmas and their proofs for the completeness of the results in this section.

**Lemma 4.11.** For the insertion channel $\text{Ins}(p)$, it holds that

$$P_{\text{run}}^{\text{ins}}(q, p) \approx \frac{q + 1}{q(q - 1)} p^2.$$  

**Proof** The probability that both channels have experienced an insertion in a given run of length $r$ is roughly $((r + 1) \cdot \frac{p}{q})^2$. Similarly to our proof of Lemma 4.5 the occurrence probability of a run of length exactly $r$ is $\left(\frac{1}{q}\right)^{r-1} \cdot \frac{q-1}{q}$. Thus, for $n$
large enough, the error probability is approximated by

\[
\sum_{r=0}^{\infty} \left( (r + 1) \cdot \frac{p}{q} \right)^2 \left( \frac{1}{q} \right)^{r-1} \cdot \frac{q - 1}{q} = \frac{p^2}{q^2} \cdot \frac{q - 1}{q} \sum_{r=0}^{\infty} (r + 1)^2 \left( \frac{1}{q} \right)^{r-1} = \frac{p^2}{q^2} \cdot \frac{q - 1}{q} \cdot \frac{q^3 (q + 1)}{(q - 1)^3} = p^2 \cdot \frac{(q + 1)}{(q - 1)^2}.
\]

The expected number of runs in a vector of length \( n \) is \( n \cdot \frac{q - 1}{q} \), thus the approximated number of insertions due to runs is

\[
n \cdot \frac{q - 1}{q} \cdot p^2 \cdot \frac{(q + 1)}{(q - 1)^2} = np^2 \cdot \frac{q + 1}{q(q - 1)},
\]

which verifies the statement in the lemma.

**Lemma 4.12.** For the insertion channel \( \text{Ins}(p) \), it holds that

\[
P_{\text{alt}}^{\text{ins}}(q, p) \approx \frac{2}{q} p^2.
\]

**Proof** Assume there are two close insertions. First insertion is in the \( i \)-th position of the first channel, and the closest insertion in the second channel is in position \( i + j \). If \( x_{[i:i+j]} \) is an alternating sequence \( ABAB \cdots \), the same output from the two channels could be received if the transmitted word is the same as \( x \) but with changing the order of the alternating sequence, that is, the symbols of the word in the positions of \([i : i + j]\) are \( BABA \cdots \). The same holds if the closest insertion in the second channel is in position \( i - j \). Therefore, the occurrence probability of this event can be approximated by

\[
2 \left( \frac{p}{q} \right)^2 \cdot \sum_{j=0}^{\infty} \frac{q - 1}{q} \cdot \frac{1}{q^{j+1}} = \frac{2}{q} \cdot p^2,
\]

where \( \frac{q - 1}{q} \cdot \frac{1}{q^{j+1}} \) is the probability that \( x_{[i:i+j]} \) is any alternating sequence and the multiplication by 2 takes into account the cases of deletion in either position \( i - j \) or \( i + j \).

**Lemma 4.13.** If there is an insertion in the first, second channel in position \( i - 1, i + j \), where \( j > 0 \), respectively, and the sequence \( x_{[i:i+j]} \) is neither a run nor
an alternating sequence, then these insertions are corrected successfully by the ML decoder.

**Proof** The output of the first channel is \( (A, x_i, x_{i+1}, x_{i+2}, \ldots, x_{i+j}) \) and the one of the second channel is \( (x_i, x_{i+1}, \ldots, x_{i+j}, B) \). Clearly, it holds that \( (x_i, x_{i+1}, x_{i+2}, \ldots, x_{i+j}) \) is a candidate for the ML decoder. Let the vector \( \hat{x}' = (x_i', x_{i+1}', x_{i+2}', \ldots, x_{i+j}') \) be another candidate for the ML decoder. The insertions occurred in positions \( i - 1 \) and \( i + j \). Hence, adding \( A' \) to the \( i \) position of \( \hat{x}' \) or \( B' \) to its \( i + j + 1 \) position will result as the two channels. Hence, it should hold that

\[
(A', x_i', x_{i+1}', x_{i+2}', \ldots, x_{i+j}') = (x_i, x_{i+1}, x_{i+2}, \ldots, x_{i+j-2}, x_{i+j-1}, x_{i+j}, B)
\]

and

\[
(x_i', x_{i+1}', x_{i+2}', \ldots, x_{i+j}', B') = (A, x_i, x_{i+1}, x_{i+2}, x_{i+3}, \ldots, x_{i+j-2}, x_{i+j-1}, x_{i+j})
\]

Therefore we get that \( x_i = x_{i+2} = x_{i+4} = \cdots \) and \( x_{i+1} = x_{i+3} = x_{i+5} = \cdots \), i.e., \( x_{[i;i+j]} \) is either the alternating sequence or a run, which results with a contradiction.

**Theorem 4.14.** The Levenshtein error rate for two insertion channels is approximated by

\[
P_{err}^{ins}(q,p) \approx P_{run}^{ins}(q,p) + P_{alt}^{ins}(q,p) + O(p^3) = \frac{3q-1}{q(q-1)}p^2 + O(p^3).
\]

**Proof** The proof follows from the above few lemmas. Note that each run error translates to an increase of the Levenshtein distance by one. On each occurrence of the alternating event the decoder chooses the correct subsequence with probability 0.5 and every error increases the Levenshtein distance by two since it translates to one insertion and one deletion. Lastly, the \( O(p^3) \) expression compensates for all other less dominant error events which introduce more than two deletions that are close to each other at least in one of the channels.

The theoretical results of Theorem 4.14 have also been verified by simulation results over words of length \( n = 500 \), which were used to create two noisy copies with a given fixed insertion probability \( p \in [0.005, 0.05] \). Then, the two copies were decoded with the ML decoder according to Claim 4.4. Lastly, we calculated and plotted in Fig. 4.4 the Levenshtein error rate as well as the error rates from runs and alternating sequences.
Figure 4.4: Levenshtein error rate by the insertion probability $p$. This simulation verifies Theorem 4.14.
Chapter 5

Reconstruction Algorithms for DNA-Storage Systems

In this chapter we present several reconstruction algorithms, crafted for DNA storage systems. Since our purpose is to solve the reconstruction problem as it is reflected in DNA-storage systems, our algorithms aim to minimize the distance between the output and the original strands. The algorithms in this work are different from most of the previously published reconstruction algorithms in several respects. Firstly, we do not require any assumption on the input. That is, the input can be arbitrary and does not necessarily belong to an error-correcting code. Secondly, our algorithms are not limited to specific cluster size, do not require any dependencies between the error probabilities, and do not assume zero errors in any specific location of the strands. Thirdly, we limit the complexity of our algorithms, so they can run with practical time on actual data from previous DNA-storage experiments. Lastly, since clusters in DNA storage systems may vary in their size and errors distributions, our algorithms are designed to minimize the distance between our output and the original strand, taking into account these errors can be corrected by the use of an error-correcting code.

5.1 Preliminaries and Problem Definition

In this chapter we use the same notations and definitions from Section 4.1. We highly recommend the reader to carefully read this section before reading the results of this chapter.
5.1.1 The DNA Reconstruction Problem

The trace reconstruction problem was first proposed in [5] and was later studied in several theoretical works; see e.g. [29, 30, 47, 51]. Under this framework, a length-\(n\) string \(x\), yields a collection of noisy copies, also called traces, \(y_1, y_2, \ldots, y_t\) where each \(y_i\) is independently obtained from \(x\) by passing through a deletion channel, under which each symbol is independently deleted with some fixed probability \(p_d\). Suppose the input string \(x\) is arbitrary. In the trace reconstruction problem, the main goal is to determine the required minimum number of i.i.d traces in order to reconstruct \(x\) with high probability. This problem has two variants: in the “worst case”, the success probability refers to all possible strings, and in the “average case” (or “random case”) the success probability is guaranteed for an input string \(x\) which is chosen uniformly at random.

The trace reconstruction problem can be extended to the model where each trace is a result of \(x\) passing through a deletion-insertion-substitution channel. Here, in addition to deletions, each symbol can be switched with some substitution probability \(p_s\), and for each \(j\), with probability \(p_i\), a symbol is inserted before the \(j\)-th symbol of \(x\). Under this setup, the goal is again to find the minimum number of channels which guarantee successful reconstruction of \(x\) with high probability.

Motivated by the storage channel of DNA and in particular the fact that different clusters can be of different sizes, this work is focused on another variation of the trace reconstruction problem, which is referred by the DNA reconstruction problem. The setup is similar to the trace reconstruction problem. A length-\(n\) string \(x\) is transmitted \(t\) times over the deletion-insertion-substitution channel and generates \(t\) traces \(y_1, y_2, \ldots, y_t\). A DNA reconstruction algorithm is a mapping \(R : (\Sigma_q)^t \to \Sigma_q\) which receives the \(t\) traces \(y_1, y_2, \ldots, y_t\) as an input and produces \(\hat{x}\), an estimation of \(x\). The goal in the DNA reconstruction problem is to minimize \(d_e(x, \hat{x})\), i.e., the edit distance between the original string and the algorithm’s estimation. When the channel of the problem is the deletion channel, the problem is referred by the deletion DNA reconstruction problem and the goal is to minimize \(d_L(x, \hat{x})\). While the main figure of merit in these two problems is the edit/Levenshtein distance, we will also be concerned with the complexity, that is, the running time of the proposed algorithms.

\footnote{Note that there are many interpretation for the deletion-insertion-substitution, most of them differs on the event when there are two errors on the same index. Our interpretation of this channel is described in Section 5.5.4}
5.2 Related Work

This section reviews the related works on the different reconstruction problems. In particular we list the reconstruction algorithms that have been used in previous DNA storage experiments and summarize some of the main theoretical results on the trace reconstruction problem.

5.2.1 Reconstruction Algorithms for DNA-Storage Systems

1. Baku et al. [5] studied the trace reconstruction problem as an abstraction and a simplification of the multiple sequence alignment problem in bioinformatics. Here the goal is to reconstruct the DNA of a common ancestor of several organisms using the genetic sequences of those organisms. They focused on the deletion case of this problem and suggested a majority-based algorithm to reconstruct the sequence, which they referred by the \textit{bitwise majority alignment (BMA) algorithm}. They aligned all traces by considering the majority vote per symbol from all traces, while maintaining pointers for each of the traces. If a certain symbol from one (or more) of the traces does not agree with the majority symbol, its pointer is not incremented and it is considered as a deletion. They showed and proved that even though this technique works locally for each symbol, its success probability is relatively high when the deletion probability is small enough.

2. Viswanathan and Swaminathan presented in [66] a BMA-based algorithm for the trace reconstruction problem under the deletion-insertion-substitution channel. Their algorithm extends the BMA algorithm so it can support also insertions and substitutions. It works iteratively on “segments” from the traces, where a segment consists of consecutive bits and its size is a fixed fraction of the trace that is given as a parameter to the algorithm. The segment of each trace is defined by its pointers. The pointers of the traces are updated in each iteration similarly as in the BMA algorithm. Each trace is classified as valid or invalid by its distance from the majority segment. Once less than $3/4$ of the traces’ segments are valid, the rest of the bits are estimated by the valid traces. Their algorithm extends [5] and improves the results from [33], where it was shown that when the number of traces is $O(\log n)$ and the deletion-insertion probability is $O\left(\frac{1}{\log^2 n}\right)$, it is possible to reconstruct a sequence with high probability. However, it assumes the
deletion and insertion probabilities are relatively small, while the substitution probability is relatively large. In practice, these probabilities vary from cluster to cluster and do not necessarily meet these assumptions.

3. Gopalan et al. [26] used the approach of the BMA algorithm from [5] and extended it to work with deletions, insertions, and substitutions to support DNA storage systems. They also considered a majority vote per symbol with some improvements. For any trace that its current symbol did not match the majority symbol, they used a “lookahead window” to look on the next 2 (or more) symbols. Then, they compared the next symbols to the majority symbols and classified it as an error accordingly. Organick et al. conducted a large scale DNA storage experiments in [48] where they successfully reconstructed their sequences using the reconstruction algorithm of Gopalan et al. [26].

4. For the case of sequencing via nanopore technology, Duda et al. [19] studied the trace reconstruction problem, while considering insertions, deletions, and substitutions. They focused on dividing the sequence into homopolymers (consecutive replicas of the same symbol), and proved that the number of copies required for accurately reconstructing a long strand is logarithmic with the strand’s length. Yazdi et al. used in [70] a similar but different approach in their DNA storage experiment. They first aligned all the strands in the cluster using the multiple sequence alignment algorithm MUSCLE [20,44]. Then, they divided each strand into homopolymers and performed majority vote to determine the length of each homopolymer separately. Their strands were designed to be balanced in their GC-content, which means that 50% of the symbols in each strands were G or C. Hence, they could perform additional majority iterations on the homopolymers’ lengths until the majority sequence was balanced in its GC-content. All of these properties guaranteed successful reconstruction of the strands and therefore they did not need to use any error-correcting code in their experiment [70].

5.2.2 Theoretical Results on the Trace Reconstruction Problem

1. Holenstein et al. [30] presented an algorithm for reconstructing a random string using polynomially many traces from the deletion channel. In their
work they assumed that the deletion probability is constant and is smaller than some threshold $\gamma$. Their work suggested a slightly different technique than [5], in the sense that they did not use standard majority voting, but a different majority scheme, where each trace vote is utilized with a probability measure of its certainty. They assumed that the last $O(\log n)$ of the input string ("anchor") can not be affected by the deletion channel, or be removed to another part of the strings. Thus, the certainty of each trace is estimated by checking if the last $O(\log n)$ bits of the trace ("anchor") match the last $O(\log n)$ of the recovered string. The threshold $\gamma$ from [30] was later estimated to be at most 0.07 by Peres and Zhai [51].

2. Peres and Zhai [51] also improved the work by Holenstein et al. [30] and the work by McGregor et al. [45]. They not only extended the range of the supported deletion probability to be $[0, 1/2)$, but also showed that a subpolynomial number of traces, more specifically $\exp(\Omega(\log^2 n))$, is sufficient for the reconstruction of a random string. Their approach includes two steps, where in the first one the strings are aligned and then, in the second step, a majority-based algorithm is invoked in order to reconstruct the sequence. The alignment step of their algorithm is quite similar to the “anchor” technique as presented in [30]. The difference is that Peres and Zhai did not restrict the “anchor” to be just the last $O(\log n)$ bits in the strings, but it could be placed at any position in the traces.

3. Holden, Pemantle, and Peres improved in [29] the upper bound on the number of traces for the random case to $\exp(\Omega(\log^{1/3} n))$ while both insertions and deletions are allowed in any probability from the range $[0, 1)$. Their algorithm consists of three ingredients: (i) A boolean test $T(w, w')$ works on pairs of sequences of the same length and indicates whether $w'$ is likely to be a result of $x$ passing through the deletion-insertion channel. (ii) An alignment procedure that creates for each of the traces $y_i$ an estimate $\tau$ for the matching between positions in $y_i$ and $x$. (iii) A bit recovery procedure that uses the aligned traces in order to estimate a bit or subsequence of bits.

4. For the worst case scenario, in [30] it is shown that $\exp(O(n^{1/2} \log n))$ traces suffice for reconstruction with high probability. This was later improved independently by both De, O’Donnell, and Severdio in [18] and by Nazarov and Peres [47] to $\exp(O(n^{1/3}))$. 

59
5. Two recent works [9, 15] studied the trace reconstruction problem for the codes setup, i.e., the transmitted is sequence not arbitrary but belongs to some code with error-correction capabilities.

6. Another related model has been studied by Kiah et al. who introduced in [35] another approach for the trace reconstruction problem, where they used profile-vectors-based coding scheme in order to reconstruct the sequence.

7. Another related problem, phrased as the sequence reconstruction problem, was also studied by Levenshtein in [39] and [40], but his approach was different. Under this paradigm he studied the minimum number of different (noisy) channels that is required in order to build a decoder that can reconstruct any transmitted sequence in the worst case. In [39], he showed that the number of channels that is required to recover any sequence has to be greater than the maximum intersection between the error balls of any two transmitted sequences. Since Levenshtein studied the worst case of this problem, the number of unique channels has to be extremely large which is not applicable for the practical setup we consider in the reconstruction of DNA strands in a DNA-based storage system.

5.3 The Deletion DNA Reconstruction Problem

This section studies the deletion DNA reconstruction problem. Assume that a cluster consists of \( t \) traces, \( y_1, y_2, \ldots, y_t \), where all of them are noisy copies of a synthesized strand. This model assumes that every strand is a sequence that is independently received by the transmission of a length-\( n \) sequence \( x \) (the synthesized strand) through a deletion channel with some fixed deletion probability \( p_d \). Our goal is to propose an efficient algorithm which returns \( \hat{x} \), an estimation of the transmitted sequence \( x \), with the intention of minimizing the Levenshtein distance between \( x \) and \( \hat{x} \). We consider both cases when \( t \) is a fixed small number and large values of \( t \).

5.3.1 An Algorithm for Small Fixed Values of \( t \)

Our approach is based on the maximum likelihood decoder over the deletion channel as presented in [54] [50]. A straightforward implementation of this approach
on a cluster of size $t$ is to compute the set of shortest common supersequences of $y_1, y_2, \ldots, y_t$, i.e., the set $\text{SCS}(y_1, y_2, \ldots, y_t)$, and then return the maximum likelihood sequence among them. This algorithm has been rigorously studied in \cite{54} and in Chapter 4 to analyze its Levenshtein error rate for $t = 2$. The method to calculate the length of the SCS commonly uses dynamic programming \cite{31} and its complexity is the product of the lengths of all sequences. Hence, even for moderate cluster sizes, e.g. $t \geq 5$, this solution will incur high complexity and impractical running times. However, for many practical values of $n$ and $p_d$, the original sequence $x$ can be found among the list of SCSs while taking less than $t$ traces or even only two of them. This fact, which we verified empirically, can drastically reduce the complexity of the ML-based algorithm. Furthermore, note that $x$ is always a common supersequence of all traces, however it is not necessarily the shortest one. Hence, our algorithm works as follows. The algorithm creates sorted sets of $r$-tuples, where each tuple consists of $r$ traces from the cluster. The $r$-tuples are sorted in a non-decreasing order according to the sum of their lengths. For each $r$-tuple $(y_{i_1}, \ldots, y_{i_r})$, the algorithm first calculates its length of the SCS, i.e., the value $\text{SCS}(y_{i_1}, \ldots, y_{i_r})$. Observe that if $\text{SCS}(y_{i_1}, \ldots, y_{i_r}) = n$ then the sequence $x$ necessarily appears in the set of SCSs of $(y_{i_1}, \ldots, y_{i_r})$, that is, $x \in \text{SCS}(y_{i_1}, \ldots, y_{i_r})$. However it is not necessarily the only sequence in $\text{SCS}(y_{i_1}, \ldots, y_{i_r})$. Hence, all is left to do is to filter the set $\text{SCS}(y_{i_1}, \ldots, y_{i_r})$ with sequences that are supersequences of all $t$ traces and finally return the maximum likelihood among them. The algorithm iterates over all possible $r$-tuples for $r = 2, 3, 4$ and if none of them succeeds, the algorithm computes all SCSs of maximal length that were observed throughout its run and returns the one that minimizes the sum of Levenshtein distances from all copies in the cluster.

In Algorithm 1 we present a pseudo-code of our solution for the deletion DNA reconstruction problem. Note that the algorithm uses another procedure which is presented in Algorithm 2 to filter the supersequences and output the maximum likelihood supersequence. The input to the algorithm is the length $n$ of the original sequence, and a cluster of $t$ traces $C$. Algorithm 1’s main loop is in Step 2; first in Step 2-a it generates the set $F$, which is a sorted set of all $r$-tuples of traces by the sum of their lengths. Then, in Step 2-b it iterates over all $r$-tuples in $F$ and checks for each $r$-tuple, $(y_{i_1}, \ldots, y_{i_r})$, if the length of their SCS, i.e., $\text{SCS}(y_{i_1}, \ldots, y_{i_r})$, equals $n$. If it is equal to $n$, it computes the set of all its SCSs, $\text{SCS}(y_{i_1}, \ldots, y_{i_r})$, and invokes Algorithm 2. Algorithm 2 checks if one or more of those SCSs are supersequences of all of the traces in the cluster, and if so it returns the maximum
likelihood among them. In case that $\text{SCS}(y_{i_1}, \ldots, y_{i_r}) < n$, the algorithm checks also if it is equal or greater than $n_{\text{max}}$, which is the longest SCS that was found so far. The algorithm saves also $C_{\text{max}}$, which is the set of all $r$-tuples such that the length of their SCS equals $n_{\text{max}}$. In Step 3, the algorithm computes $S_{\text{max}} = \bigcup_{c \in C_{\text{max}}} \text{SCS}(c)$, which is the union of sets of SCSs of the $r$-tuples that the length of their SCS was $n_{\text{max}}$. In Step 4, the algorithm invokes again Algorithm 2 to check if $S_{\text{max}}$ includes supersequences of all traces in $C$ and returns the maximum likelihood among them. If none of the sequences in $S_{\text{max}}$ is a supersequence of all traces in $C$, the algorithm returns in Step 5 the sequence which minimizes the sum of Levenshtein distances to all the traces in $C$.

5.3.2 Simulations

We evaluated the accuracy and efficiency of Algorithm 1 by the following simulations. These simulations were tested over sequences of length $n = 200$, clusters of size $4 \leq t \leq 10$, and deletion probability $p$ in the range $[0.01, 0.10]$. Each simulation consisted of 100,000 randomly generated clusters. Furthermore, we had another set of simulations for $n = 100$ with deletion probability $p$ in the range $[0.11, 0.20]$ and clusters of size $4 \leq t \leq 10$. Each simulation for these values of $p, n$, and $t$ included 10,000 randomly selected clusters. We calculated the Levenshtein error rate (LER) of the decoded output sequence as well as the average decoding success probability (referred as the success rate). We also calculated the $k$-error success rate, which is defined as the fraction of clusters where the Levenshtein distance between the algorithm’s output sequence and the original sequence was at most $k$. Note that for $k = 0$, this is equivalent to calculate the success rate. We also calculated the minimal $k$ for which its $k$-error success rate is at least $q$, and denote this value of $k$ by $k_{q,\text{succ}}$. Note that for $q = 1$ this value determines the minimal number of Levenshtein errors that an error-correcting code must correct in order to fully decode the original sequences using Algorithm 1 with an error-correcting code. In addition, each cluster was also reconstructed using the BMA algorithm [5].

Figure 5.1 presents the LER as computed in our simulations of Algorithm 1 and the BMA algorithm for clusters of sizes $t = 7$ and $t = 10$. We also added the trivial lower bound of $p^t$ on the LER [54, 60]. This bound corresponds to the case when the same symbol is deleted in all of the traces. In this case, this symbol will not appear in the list of SCSs of any possible $r$-tuple or even the entire cluster since it cannot be recovered. Hence, it is not possible to recover its value and thus it will
Algorithm 1 ML-SCS Reconstruction

Input:
- Cluster $C$ of $t$ noisy traces: $y_1, y_2, \ldots, y_t$ sorted by their lengths from the longest to the shortest.
- Design length $= n$.

Output: $\hat{x}$ - Estimation of the original sequence.

1. $\hat{x} = \epsilon$, $n_{\text{max}} = 0$, $C_{\text{max}} = \emptyset$.

2. for $r = 2, 3, 4$ do
   (a) Denote $F = \{ c_i^{(r)} = (y_{i_1}, y_{i_2}, \ldots, y_{i_r}) | 1 \leq i \leq \binom{t}{r}, 1 \leq i_1 < i_2 < \cdots < i_r \leq t \}$ the set of all $r$-tuples from $C$, sorted by non-decreasing order of the sum of the lengths of the copies in each tuple.
   (b) for $i = 1, 2, \ldots, \binom{t}{r}$ do
      if $\text{SCS}(c_i^{(r)}) = n$ then
        $S = \text{SCS}(c_i^{(r)})$
        $\hat{x} = \text{ML-Supersequence}(S, C)$
        if $\hat{x} \neq \epsilon$ then
          return $\hat{x}$
        end if
      end if
      else
      if $\text{SCS}(c_i^{(r)}) > n_{\text{max}}$ then
        $n_{\text{max}} = \text{SCS}(c_i^{(r)})$
        $C_{\text{max}} = \{ c_i^{(r)} \}$
      end if
      if $\text{SCS}(c_i^{(r)}) = n_{\text{max}}$ then
        $C_{\text{max}} = C_{\text{max}} \cup \{ c_i^{(r)} \}$
      end if
    end if
  end for
end for
3. Compute $S_{\text{max}} = \bigcup_{c \in C_{\text{max}}} \text{SCS}(c)$, the union of all SCS of $c_i^{(r)} \in C_{\text{max}}$.
4. $\hat{x} = \text{ML-Supersequence}(S_{\text{max}}, C)$
5. if $\hat{x} \neq \epsilon$ then
    return $\hat{x}$
else
    Return the sequence from $S_{\text{max}}$ that has the minimum sum of Levenshtein distance to the copies in the cluster.
end if
Algorithm 2 ML-Supersequence

Input:
- Cluster $C$ of $t$ noisy traces: $y_1, y_2, \ldots, y_t$.
- $S=\{s_1, s_2, \ldots, s_k\}$, a set of $k$ candidates.

Output:
Maximum likelihood candidate of $S$, that is supersequence of all copies from the cluster. If it is not exists, the algorithm returns $\epsilon$.

1. Filter $S$ so it contains only sequences which are supersequence of all traces from the cluster.
2. if $S \neq \emptyset$ then
   Return the maximum likelihood sequence from $S$ with a respect to cluster $C$.
end if
3. Return $\epsilon$.

be deleted also in the output of the ML decoder.

In order to simulate also high deletion probabilities, we simulated 1000 clusters of length $n = 100$ with cluster size $t$ between 4 and 10, while the deletion probability was $p = 0.25$. Figure 5.2 a) presents the $k$-error success rate of this simulation and Figure 5.2 b) presents the values of $k_{1.\text{succ}}$ and $k_{0.99.\text{succ}}$ by the cluster size in the simulation.

5.3.3 Large Cluster

In case the cluster is of larger size, for example in the order of $\Theta(n)$, we present in Algorithm 3 a variation of Algorithm 1 for large clusters. In this case, since the cluster is large, the probability to find a pair, triplet, or quadruplet of traces that their set of SCSs contains the original sequence $x$ is very high, if not even 1. In fact, in all of our simulations, which we will elaborate below in this section, we were always able to successfully decode the original sequence with no errors even when the deletion probability was as high as 0.2. Hence, our main goal in this part is to decrease the runtime of Algorithm 1 while preserving the success rate to be 1. Algorithm 3 keeps the same structure of Algorithm 1, however, it performs two
filters on the cluster in order to reduce the computation time.

The complexity of finding the length of the SCS of some set of \( r \) traces is the multiplication of their lengths, i.e., \( \Theta(n^r) \) \[^3^1\]. Therefore, the complexity of finding the length of the SCS of a pair of traces is \( \Theta(n^2) \), while there are \( \Theta(n^2) \) pairs of traces (assuming the cluster size is \( \Theta(n) \)). Therefore, in this case, calculating the length of the SCS of each pair of traces before considering some triplets is not necessarily the right strategy when our goal is to optimize the algorithm’s running time. Hence, in Algorithm \[^3\] we focused on filtering the traces in the cluster in order to check only a subset of the traces which are more likely to succeed and produce the correct sequence.

To define the filtering criteria for Algorithm \[^3\] we simulated Algorithm \[^1\] on large clusters. The length of original sequence \( x \) was \( n = 200 \) and the cluster size was \( t = \frac{n}{2} = 100 \). We generated 10,000 clusters of size \( t \), where the deletion probability \( p \) was in the range \([0.01, 0.15]\). The success rate of all the simulations was 1. We evaluated the percentage of clusters that the first \( r \)-tuple to have an SCS of size \( n \) was consisted of the longest 20% traces in the cluster. We observed that when the deletion probability was at most 0.07, in all of the clusters the first \( r \)-tuple of traces that had an SCS of size \( n \) consisted from the longest 20% traces in the cluster. For deletion probabilities between 0.08 and 0.11 these percentages ranged between 94.76% and 99.98%, while for \( p = 0.15 \) this percentage was 60.88%. Therefore, by filtering the longest 20% traces, it is enough to check only \( \binom{20}{2} \) pairs instead of \( \binom{100}{2} \) pairs in order to succeed and still reach the successful pair. The results of these simulations are depicted in Figure 5.4(a).

This observation led us to the first filter in Algorithm \[^3\] where we picked the longest 20% traces of the cluster. The second filter computes a cost function (in linear time complexity), to be explained below, on a given \( r \)-tuple of traces in order to evaluate if the traces in this \( r \)-tuple are likely to have a SCS of length \( n \). Thus, the algorithm skips on the SCS computation of \( r \)-tuples that are less likely to have a SCS of length \( n \). First, before performing the first filter, the algorithm calculates the average length of the traces in the cluster and uses it to estimate the deletion probability \( p \). Then, if \( p > 0.1 \), the algorithm calculates the cost function on every \( r \)-tuple and checks if it higher than some fixed threshold. This threshold depends on the estimated value of \( p \) and the cost function is based on a characterization of the sequences, as will be described in Section 5.3.3.
An Algorithm for Large Values of $t$

In this section we present Algorithm 3. We list here the steps that are different from Algorithm 1. In Step 2 the algorithm estimates the deletion probability in the cluster by checking the average length of the traces $n'$ and then calculates $p = 1 - \frac{n'}{n}$. In Step 3, the algorithm filters the cluster so it contains only the longest 20% traces. The last difference between Algorithm 3 and Algorithm 1 can be found in Step 4-b. In this step, before the computation of the SCS of a given $r$-tuple of traces, the algorithm computes the $k$-mer cost function (for $k$-mers of size $k = 2$) and checks if it is larger than the threshold $T_p$.

We evaluated the performance of Algorithm 3 and verified our filters by simulations. Each simulation consisted of 10,000 clusters of size $t = 100$, the length of the original strand was $n = 200$, and the deletion probability $p$ was in the range $[0.01, 0.2]$. Algorithm 3 reconstructed the exact sequence $x$ in all of the tested clusters. A comparison between the runtime of Algorithm 1 and Algorithm 3 can be found in Figure 5.4(b). Note that we did not compare the running time with the BMA algorithm since its success rate was significantly lower, for example when the deletion probability was 15%, its success rate was roughly 0.46.

The $k$-mer Distance and the $k$-mer Cost Function

The $k$-mer vector of a sequence $y$, denoted by $k$-mer($y$), is a vector that counts the frequency in $y$ of each subsequence of length $k$ (k-mer). The frequencies are ordered in a lexicographical order of their corresponding k-mers. For example for a given sequence $y = "ACCTCC"$ and $k = 2$, its k-mer vector is $k$-mer($y$) = 010002010000101, according to the following calculation of the frequencies \{AA : 0, AC : 1, AG : 0, AT : 0, CA : 0, CC : 2, CG : 0, CT : 1, GA : 0, GC : 0, GG : 0, GT : 0, TA : 0, TC : 1, TG : 0, TT : 1\}. We define the $k$-mer distance between two sequence $y_1$ and $y_2$ as the $L_1$ distance between their k-mer vectors. The $k$-mer distance is denoted by $d_{k\text{-}}\text{mer}(y_1, y_2)$.

$$d_{k\text{-}}\text{mer}(y_1, y_2) = ||y_1 - y_2||_1$$

For a given set of $r$ sequences $Y = \{y_1, y_2, \ldots, y_r\}$, we define its $k$-mer cost function, which is denoted by $c_{k\text{-}}\text{mer}(y_1, y_2, \ldots, y_r)$, as the sum of the $k$-mer distance
Algorithm 3 ML-SCS Reconstruction for Large Clusters

Input:
- Cluster $C$ of $t = \Theta(n)$ noisy traces: $y_1, y_2, \ldots, y_t$ sorted by their lengths in a non-decreasing order.
- Design length $= n$.

Output: $\hat{x}$ - Estimation of the original sequence.

1. $\hat{x} = \epsilon$, $n_{\text{max}} = 0$, $C_{\text{max}} = \emptyset$.
2. Compute $n'$ the mean length of the traces in $C$, and define $p = 1 - \frac{n'}{n}$.
3. Filter traces from $C$ so it contains only the $t' = 0.2t$ first traces in the cluster.
4. for $r = 2, 3, 4$ do
   (a) Denote $F = \{e_i^{(r)} = (y_{i_1}, y_{i_2}, \ldots, y_{i_r})|1 \leq i \leq \binom{t'}{r}, 1 \leq i_1 < i_2 < \cdots < i_r \leq t'\}$ the set of all $r$-tuples from $C$, sorted by non-decreasing order of the sum of the lengths of the copies in each tuple.
   (b) for $i = 1, 2, \ldots, \binom{t'}{r}$ do
      if $p > 0.1$ and $c_{\text{mer}}(e_i^{(r)}) > 0.25np(2k - 1)$ then
         /* k-mer size $k = 2$ */
         if SCS($e_i^{(r)}$) = $n$ then
            $S = \text{SCS}(e_i^{(r)})$
            $\hat{x} = \text{ML-Supersequence}(S, C)$
            if $\hat{x} \neq \epsilon$ then
               return $\hat{x}$
            end if
         end if
         if SCS($e_i^{(r)}$) > $n_{\text{max}}$ then
            $n_{\text{max}} = \text{SCS}(e_i^{(r)})$
            $C_{\text{max}} = \{e_i^{(r)}\}$
         end if
      end if
   end for
   (c) end for
5. Compute $S_{\text{max}} = \bigcup_{e \in C_{\text{max}}} \text{SCS}(e)$, the union of all SCS of $e_i^{(r)} \in C_{\text{max}}$.
6. $\hat{x} = \text{ML-Supersequence}(S_{\text{max}}, C)$
7. if $\hat{x} \neq \epsilon$ then
       return $\hat{x}$
    else
       return $\arg \min_{s \in S_{\text{max}}} \sum_{y_i \in C} (d_L(s, y_i))$
    end if
of each pair of sequences in $Y$. That is,

$$c_{k\text{-mer}}(y_1, y_2, \ldots, y_r) = \sum_{1 \leq i < j \leq r} d_{k\text{-mer}}(y_i, y_j)$$

Observe that the $k$-mer distance between a sequence $x$ and a trace $y_i$ which results from $x$ by one deletion is at most $2k - 1$. Every deleted symbol in $x$ decreases the value of at most $k$ entries in $k\text{-mer}(x)$ and increases the number of at most $k - 1$ of the entries. Hence, each deletion increases the $k$-mer distance by at most $2k - 1$, which means that an upper bound on the $k$-mer distance between the original strand $x$ and a trace $y_i$ with $np$ deletions is $np(2k - 1)$. However, when comparing the $k$-mer distance of two traces, $y_1$ and $y_2$, with more than one deletion, the $k$-mer distance can also decrease. An example of such a case is depicted in Figure 3.

Combining these two observations, Algorithm 3 estimates if two traces have relatively large Levenshtein distance. If these traces have large Levenshtein distance, it is more likely that both of them will have an SCS of length $n$. Hence, the algorithm checks if the $k$-mer distance is larger than the threshold $T_p = 0.25np(2k - 1)$ and continues to compute the SCS, only if the condition holds. A similar computation is done for tuples with more than two traces. We use the value of 0.25 in the threshold to consider the cases where the $k$-mer distance decreases as depicted in Figure 5.3. We selected this value after simulating other values as well, reaching the best result with 0.25. An optimization of this value can be done in further research.

### 5.4 The DNA Reconstruction Problem

This section studies the DNA reconstruction problem. Assume that a cluster consists of $t$ traces, $y_1, y_2, \ldots, y_t$, where all of them are noisy copies of a synthesized strand. This model assumes that every trace is a sequence that is independently received by the transmission of a length-$n$ sequence $x$ (the synthesized strand) through a deletion-insertion-substitution channel with some fixed probability $p_d$ for deletion, $p_i$ for insertion, and $p_s$ for substitution. Our goal is to propose an efficient algorithm which returns $\hat{x}$, an estimation of the transmitted sequence $x$, with the intention of minimizing the edit distance between $x$ and $\hat{x}$. In our simulations, we consider several values of $t$ and a wide range of error probabilities as well as data from previous DNA storage experiments.

Before we present the algorithms, we list here several more notations and def-
Algorithm 4 returns as output the algorithm perform these steps on \( C \) the most frequent symbol in this entry, and saves it in the of '-' in at most LCS for all \( y \). Each of the anchor symbols is located in a specific index of the rest of the symbols in the mapping vector \( V \) each of the \( t \) \( y \) and \( c \) receives \( j \) appears as the \( |lcs| \) size \( lcs \) of the symbol '-' in insertion. Second, the algorithm computes farthest to \( n \) \( y \), the algorithm performs a majority vote on the \( i \) \( x \) that, in the case where the \( x \) in \( y \), the algorithm writes the symbol symbol appears as the \( j \)-th symbol in \( x \), with respect to the error vector \( EV(y, x) \). Note that, in the case where the \( i \)-th symbol in \( y \) was classified as a deleted symbol in \( EV(y, x) \), \( EV(y, x)(i) = ? \). This mapping can also be represented as a vector of size \( |y| \), where the \( i \)-th entry in this vector is \( EV(y, x)(i) \). The reversed cluster of a cluster \( C \), denoted by \( C^{bck} \), consists of the traces in \( C \) where each one of them is reversed.

### 5.4.1 The LCS-Anchor Algorithm

In this section we present Algorithm 4 the LCS-anchor algorithm. The algorithm receives \( C \), a cluster of traces sorted by their lengths, from closest to \( n \) to the farthest to \( n \). First, the algorithm initializes \( \xhat \) and \( \xhat^{bck} \) as a length-\( n \) sequence of the symbol '-' . Second, the algorithm computes \( lcs \) one of the LCSs of \( y_1 \) and \( y_2 \), the two traces in the cluster which their length is closest to \( n \). Then, for each of the \( t \) traces in the cluster, \( y_k \), the algorithms compute \( EV(y_k, lcs) \), and the mapping vector \( V_{EV}(lcs, y_k) \). For \( 1 \leq i \leq |lcs| \) the algorithm performs a majority vote on the \( i \)-th entries of the \( t \) mapping vectors. If the majority is \( j \neq ? \), the algorithm writes the symbol \( lcs(i) \) in the \( j \)-th index of \( \xhat \). If \( j = ? \) the symbol \( lcs(i) \) is omitted, and it is not written in \( \xhat \). At this point, the Algorithm 4 estimated at most LCS(\( y_1, y_2 \)) symbols in \( \xhat \), these symbol serves as “anchor” symbols in the estimated string. Each of the anchor symbols is located in a specific index of \( \xhat \), and the rest of the symbols in \( \xhat \) are ‘-’. Note that the anchor symbol are not necessarily placed in consecutive indices of \( \xhat \). In the following steps, Algorithm 4 computes for all \( y_k \in C \), the vectors \( EV(\xhat, y_k) \) and \( V_{EV}(\xhat, y_k) \). Then, for each \( h \), an entry of ‘-’ in \( \xhat \), the algorithm performs a majority vote on \( y_k(V_{EV}(\xhat, y_k)(h)) \), to find the most frequent symbol in this entry, and saves it in the \( h \)-th entry of \( \xhat \). Lastly, the algorithm perform these steps on \( C^{bck} \) and saves the resulted sequence in \( \xhat^{bck} \). Algorithm 4 returns as output \( \xhat_{1,2,\ldots,|\frac{n}{2}|}^{bck} x_{\frac{n}{2}+1,\ldots,|\frac{n}{2}|} \).
Algorithm 4 LCS-Anchor

Input:
- Cluster $C$ of $t$ noisy traces: $y_1, y_2, \ldots, y_t$, sorted by their lengths from closest to the farthest to $n$.
- Design length $= n$

Output:
- $\hat{x}$ - Estimation of the original sequence

1. Initialize $\hat{x}$ and $\hat{x}^{\text{bck}}$ as a length-$n$ sequence of the symbol ‘-’.
2. Calculate $lcs$ one of the LCSs of $y_1, y_2$.
3. for all $y_k \in C$ do
   (a) Compute $EV(y_k, lcs)$.
   (b) Compute $V_{EV}(lcs, y_k)$.
end for
4. for $1 \leq i \leq |lcs|$ do
   (a) $j = 0$
   (b) For $1 \leq k \leq t$, perform a majority vote on $V_{EV}(lcs, y_k)(i)$ and save it in $j$.
   (c) If $j \neq \text{?}$, $\hat{x}(j) = lcs(i)$
end for
5. for all $y_k \in C$ do
   (a) Compute $EV(y_k, \hat{x})$.
   (b) Compute $V_{EV}(\hat{x}, y_k)$.
end for
6. for all $h$ s.t. $\hat{x}(h) = '-'$ do
   (a) For $1 \leq k \leq t$, perform a majority vote on $y_k(V_{EV}(\hat{x}, y_k)(h))$ and save it in $m$.
   (b) $\hat{x}(h) = m$.
end for
7. Repeat Steps 3-6 for $C^R$ and save the results in $\hat{x}^{\text{bck}}$.
8. Return $\hat{x}_{1,2,\ldots,\lceil \frac{n}{2} \rceil}, \hat{x}^{\text{bck}}_{1,2,\ldots,\lfloor \frac{n}{2} \rfloor}$
5.4.2 The Iterative Reconstruction Algorithm

In this section we present Algorithm 5. The algorithm receives a cluster of \( t \) traces \( C \) and the design length \( n \). Algorithm 5 uses several methods to revise the traces from the cluster and to generate from the revised traces a multiset of candidates. Then, Algorithm 5 returns the candidate that is most likely to be the original sequence \( x \). The methods used to revise the traces are described in this section as Algorithm 6 and Algorithm 7. Algorithm 5 invokes Algorithm 6 and Algorithm 7 on the cluster in two different procedures as described in Algorithm 8 and Algorithm 9.

The first method is described in Algorithm 6. The algorithm receives \( C \), a cluster of \( t \) traces, the design length \( n \), and \( y_k \), a trace from the cluster. Algorithm 6 calculates for every \( 1 \leq h \leq t, h \neq k \), the vector \( EV(y_k, y_h) \). Then, the algorithm performs a majority vote in each index on these vectors and creates \( S \), which is a vector of edit operations. Lastly, Algorithm 6 performs the edit operations on \( y_k \), and returns it as an output for Algorithm 8 and Algorithm 9. Algorithm 6 is used as a procedure in Algorithm 8 and Algorithm 9 to correct substitution and insertion errors of the traces in the cluster.

The second method is described in Algorithm 7. Similarly to Algorithm 6, Algorithm 7 receives \( C \), a cluster of \( t \) traces, the design length \( n \), and \( y_k \), a trace from the cluster. Algorithm 7 uses similar patterns (defined in Section 5.4.2) on each pair of traces and creates a weighted graph from them. Each vertex of the graph represents a pattern, and an edge connects patterns with identical prefix and suffix. The weight on each edge represents the frequency of the incoming pattern, the number of pairs of traces in the cluster that have this pattern in their sequences. Algorithm 7 is described in detail in Section 5.4.2. Algorithm 7 is used as a procedure in Algorithm 8 and Algorithm 9 to correct deletion errors in the traces in the cluster.

Algorithm 8 receives a cluster of \( t \) traces \( C \) and the design length \( n \). Algorithm 8 performs \( k \) cycles, where in each cycle it iterates over all the traces in the cluster. For each trace \( y_k \), it first uses Algorithm 6 to correct substitution errors, then it uses Algorithm 7 to correct deletion errors, and lastly, it uses Algorithm 6 to correct insertion errors. When it finishes iterating over the traces in the cluster, Algorithm 8 updates the cluster with all the revised traces and continues to the next cycle. At the end, Algorithm 8 performs the same procedure on \( C^R \). Algorithm 8 returns a multiset of all the revised traces.
Algorithm 9 also receives a cluster of \( t \) traces \( C \) and the design length \( n \). Algorithm 9 uses the same procedures as Algorithm 8. However, in each cycle, it first corrects substitution in all of the traces in the cluster using algorithm 6, then it invokes algorithm 7 on each trace to correct deletions, and finally invokes Algorithm 6 to correct insertions. Similarly to Algorithm 8, Algorithm 9 performs the same operations also on \( C^R \) and returns a multiset of the results.

Algorithm 5 invokes Algorithms 8 and 9, with \( k = 2 \) cycles and combines the resulted multisets to the multiset \( S \). If one or more sequences of length \( n \) exists in the multiset \( S \), it returns the one that minimizes the sum of edit distances to the traces in the cluster. Otherwise, it checks if there are sequences of length \( n - 1 \) or \( n + 1 \) in \( S \), and returns the most frequent among them. If such a sequence does not exist, it returns the first sequence in \( S \).

\[ \text{Algorithm 5 Iterative Reconstruction} \]

**Input:**
- Cluster \( C \) of \( t \) noisy traces: \( y_1, y_2, \ldots, y_t \).
- Design length = \( n \).

**Output:**
- \( \hat{x} \) - Estimation of the original sequence.

1. \( G = \emptyset \)
2. Use Algorithm 8 and Algorithm 9 with \( C, n, k = 2 \) as parameters, to compute a multiset of candidates. Save the candidates and their frequencies in it in \( S \).
3. If \( S \) has one or more sequence of length \( n \), return one that minimizes the sum of edit distance to the traces in \( C \).
4. If \( S \) has one or more sequences of length \( n - 1 \) or \( n + 1 \), return the sequence is most frequent in the multiset \( S \) (if there is more than one choose randomly).
5. Return the first sequence in \( S \).

The Pattern Path Algorithm

In this section we present Algorithm 7, the Pattern-Path algorithm. Algorithm 7 is being used to correct deletion errors. Denote by \( w \) an arbitrary LCS sequence
of \(x\) and \(y\) of length \(\ell\). The sequence \(w\) is a subsequence of \(x\), hence, all of its \(\ell\) symbols appear in some indices of \(x^t\) and assume these indices are given by \(i_1^t \leq i_2^t \leq \cdots \leq i_r^t\). For a sequence \(x\), and its subsequence \(w\) we define the embedding sequence of \(w\) in \(x\), denoted by \(u_{x,w}\) as a sequence of length \(|x|\) where for \(1 \leq j \leq \ell\), \(u_{x,w}(i_j^t)\) equals to \(x(i_j^t)\) and otherwise it equals to \(?\).

The gap of \(x\), \(y\) and their LCS sequence \(w\) in index \(1 \leq j \leq |x|\) with respect to \(u_{x,w}\), denoted by \(\text{gap}_{u_{x,w},y}(j)\), is defined as the sequence \(y(\ell_{j-1}^{y} + 1: \ell_j^{y} - 1)\), which is the sequence between the appearances of the \(j\)-th and the \((j-1)\)-th symbols of \(w\) in \(y\). Note that since \(i_j^y\) can be equal to \(i_{j-1}^y + 1\), \(\text{gap}_{u_{x,w},y}(j)\) can be an empty sequence. In case the \(j\)-th symbol in \(u_{x,w}\) equals \(?\) then \(\text{gap}_{u_{x,w},y}(j)\) is an empty sequence.

The pattern of \(x\) and \(y\) with respect to the LCS sequence \(w\), an embedding sequence \(u_{x,w}\), an index \(i\) and a length \(m\), denoted by \(Ptn(x, y, w, u_{x,w}, i, m)\), is defined as:

\[
Ptn(x, y, w, u_{x,w}, i, m) \equiv (u_{x,w}(i-1), \text{gap}_{u_{x,w},y}(i), u_{x,w}(i), \text{gap}_{u_{x,w},y}(i+1), \ldots, \text{gap}_{u_{x,w},y}(i+m-2), u_{x,w}(i+m-2)).
\]

We also define the prefix and suffix of a pattern \(Ptn(x, y, w, u_{x,w}, i, m)\)

\[
\text{Prefix}(Ptn(x, y, w, u_{x,w}, i, m)) \equiv (u_{x,w}(i-1), \text{gap}_{u_{x,w},y}(i), u_{x,w}(i), \ldots, \text{gap}_{u_{x,w},y}(i+m-3), u_{x,w}(i+m-3)),
\]

\[
\text{Suffix}(Ptn(x, y, w, u_{x,w}, i, m)) \equiv (u_{x,w}(i), \text{gap}_{u_{x,w},y}(i+1), \ldots, \text{gap}_{u_{x,w},y}(i+m-2), u_{x,w}(i+m-2)).
\]

Finally, we define

\[
P(x, y, w, u_{x,w}, m) \equiv \{Ptn(x, y, w, u_{x,w}, i, m) : 1 \leq i \leq |x|\}.
\]

Algorithm\footnote{Algorithm 7} receives a cluster \(C\) of \(t\) traces and one of the traces in the cluster \(y_k\). First, the algorithm initializes \(L[y_k]\), which is a set of \(|y_k|\) empty lists. For \(1 \leq i \leq |y_k|\), the \(i\)-th list of \(L[y_k]\) is denoted by \(L[y_k]_i\). Algorithm\footnote{Algorithm 7} pairs \(y_k\) with each of the other traces in \(C\). For each pair of traces, \(y_k\) and \(y_h\), Algorithm\footnote{Algorithm 7} computes an arbitrary LCS sequence \(w\), and an arbitrary embedding sequence \(u_{y_k,w}\). Then it uses \(w\) and \(u_{y_k,w}\) to computes \(P(y_k, y_h, w, u_{y_k,w}, m)\). For \(1 \leq i \leq |y_k|\), the

\footnote{A subsequence can have more than one set of such indices, while the number of such sets is defined as the embedding number \cite{21}. In our algorithm, we chose one of these sets arbitrarily.}
Algorithm saves $Ptn(x, y, w, u_{x,w}, i, m)$ in $L[y_k]$. 

Then, Algorithm 7 builds the pattern graph $G_{pat} = (V, E)$, which is a directed acyclic graph, and is defined as follows.

1. $V = \{((Ptn(y_k, y_h, w, u_{x,w}, i, m), i) : 1 \leq h \leq t, 1 \leq i \leq |y_k|) \cup \{S, U\}\}$.
   
   The vertices are pairs of pattern and their index. Note that the same pattern can appear in several vertices with different indices $i$. The value $|V|$ equals to the number of distinct pattern-index pairs.

2. $E = \{e = (v, u) : v = ((Ptn(x, y, w, u_{x,w}, i, m), i), u = ((Ptn(x, y, w, u_{x,w}, j, p), j), j = i + 1, \text{Suffix}(Ptn(x, y, w, u_{x,w}, i, m)) = \text{Prefix}(Ptn(x, y, w, u_{x,w}, j, m))\}$. 
   
   That is, an edge $e$ exists in $E$ if and only if:
   
   (a) $j = i + 1$
   
   (b) $\text{Suffix}(Ptn(x, y, w, u_{x,w}, i, p)) = \text{Prefix}(Ptn(x, y, w, u_{x,w}, j, p))$

3. The weights of the edges are defined by $w : E \rightarrow \mathbb{N}$ as follows:
   
   For $e = (v, u)$, where $u = (Ptn(x, y, w, u_{x,w}, j, m), j)$, it holds that
   
   $$w(e) = |\{Ptn \in L[y_k]_j : Ptn = (Ptn(x, y, w, u_{x,w}, j, m)\}|,$$
   
   which is the number of appearances of $(Ptn(x, y, w, u_{x,w}, j, m))$ in $L[y_k]_j$.

4. The vertex $S$ which does not correspond to any pattern, is connected to all vertices of the first index. The weight of these edges is the number of appearances of the incoming vertex pattern.

5. The vertex $U$ has incoming edges from all vertices of last index and the weight of each edge is zero.

Algorithm 7 finds a longest path from $S$ to $U$ in the graph. This path induces a sequence, denoted by $\hat{y}_k$, that consists of patterns of $y_k$ where some of them includes gaps. The algorithm returns $\hat{y}_k$, which is a revised version of $y_k$ while adding to the original sequence of $y_k$ the gaps that inherited from the longest path vertices.
Algorithm 6 Error Vectors Majority

Input:
- Cluster C of t noisy traces: \( y_1, y_2, \ldots, y_t \).
- Design length = n
- \( y_i \in C \) - a copy from the cluster.

Output:
- \( \hat{x} \) - a revised version of \( y_i \), an estimation of \( y_i \) with less substitution and insertion errors.

1. \( S = "", \) an empty vector.
2. \textbf{for} \( y_k \in C, \ j \neq i \text{ do} \)
   (a) Compute \( EV(y_i, y_k) \).
\textbf{end for}
3. \textbf{for} \( 1 \leq j \leq |y_i| + 1 \text{ do} \)
   (a) Set \( S(j) \) to be the operation that appeared in the \( j \)-th entry of most of the \( EV \) that computed in Step 2
\textbf{end for}
4. Perform the operations from the vector \( S \) on \( y_i \) and save the resulted sequence in \( \hat{x} \).
5. Return \( \hat{x} \).
Algorithm 7 Pattern-Path

Input:

- Cluster \( C \) of \( t \) noisy traces: \( y_1, y_2, \ldots, y_t \).
- Design length = \( n \)
- \( y_k \in C \) - a copy from cluster \( C \).

Output:

- \( \hat{y}_k \) - a revised version of \( y_k \). The sequence \( \hat{y}_k \) consists of \( y_k \)'s original symbols and also includes some additional symbols, which are estimations of the symbols deleted from \( y_k \).

1. \( L[y_k] = \{ L[y_k]_1, \ldots, L[y_k]|y_k| \} \), a list of \( |y_k| \) empty lists, where each represents the list of patterns before the symbol \( i \) in \( y_k \), where the last list represents symbols before the end of the sequence.

2. /*In this stage we pair \( y_k \) with all the copies from the cluster, create list \( L[y_k] \) of \( |y_k| \) lists of patterns of symbol \( i \) and their frequencies*/

   for \( y_h \in C \) do
      (a) Compute \( w \) an LCS sequence of \( y_k, y_h \).
      (b) Compute \( u_{y_k,w} \) an embedding sequence for \( y_k \) and \( w \).
      (c) Computes \( P(y_k, y_h, w, u_{y_k,w}, m = 3) \).
      (d) For each \( 1 \leq i \leq y_k \) add to \( L[y_h]_i \) the pattern \( P(y_k, y_h, w, u_{y_k,w}, i, 3) \).
   end for

3. Build \( G_{pat} = (V, E) \) - the pattern graph.

4. Find the longest path from the source vertex \( S \) in \( G_{pat} \).

5. Let \( \hat{y}_k \) bet the sequence that inherited from the patterns of the vertices of the longest path.

6. Return \( \hat{y}_k \).
Algorithm 8 Iterative Reconstruction - Horizontal

Input:
- Cluster $C$ of $t$ noisy traces: $y_1, y_2, \ldots, y_t$.
- Design length = $n$.

Output:
- $S = \{s_1, s_2, \ldots, s_p\}$, a multiset of $p$ candidates, that estimate the original sequence of the cluster.

1. $S = \emptyset$, $C_{\text{orig}} = C$
2. for $j = 1, 2, \ldots, k$ do
   (a) $C_{\text{tmp}} = \emptyset$
   (b) for $y_i \in C_{\text{orig}}$ do
        i. Perform Algorithm 6 on $y_i$ to correct substitution errors.
        ii. Perform Algorithm 7 on $y_i$ to correct deletion errors.
        iii. Perform Algorithm 8 on $y_i$ to correct insertion errors.
        iv. $C_{\text{tmp}} = C_{\text{tmp}} \cup \{y_i\}$.
   (c) end for
   (d) $C = C_{\text{tmp}}$.
end for
3. $S = S \cup C$.
4. Set $C_{\text{orig}} = C_{\text{orig}}^R$ and repeat Steps 2-3 on $C_{\text{orig}}^R$. Add the results to $S$.  

77
**Algorithm 9** Iterative Reconstruction - Vertical

**Input:**
- Cluster $C$ of $t$ noisy traces: $y_1, y_2, \ldots, y_t$.
- Design length = $n$.

**Output:**
- $S = \{s_1, s_2, \ldots, s_p\}$, a multiset of $p$ candidates, sequences that estimates the original sequence of the cluster.

1. $S = \emptyset$, $C_{\text{orig}} = C$
2. for $j = 1, 2, \ldots, k$ do
   (a) $C_{\text{imp}} = \emptyset$
   (b) for $y_i \in C$ do
      i. Perform Algorithm 6 on $y_i$ to correct substitutions.
      ii. $C_{\text{imp}} = C_{\text{imp}} \cup \{y_i\}$.
   (c) end for
   (d) $C = C_{\text{imp}}$
   (e) $C_{\text{imp}} = \emptyset$
   (f) for $y_i \in C$ do
      i. Perform Algorithm 7 on $y_i$ to correct deletions.
      ii. $C_{\text{imp}} = C_{\text{imp}} \cup \{y_i\}$.
   (g) end for
   (h) $C = C_{\text{imp}}$
   (i) $C_{\text{imp}} = \emptyset$
   (j) for $y_i \in C$ do
      i. Perform Algorithm 6 on $y_i$ to correct insertions.
      ii. $C_{\text{imp}} = C_{\text{imp}} \cup \{y_i\}$.
   (k) end for
   (l) $C = C_{\text{imp}}$
end for
3. $S = S \cup C$
4. Set $C_{\text{orig}} = C_{\text{orig}}^R$ and repeat Steps 2-3 on $C_{\text{orig}}^R$. Add the results to $S$. 

78
Algorithm 7 - Example

We present here a short example of Algorithm 7. Figure 5.5 presents the input of Algorithm 7 and includes the original strand \( x \) and the 5 traces \( y_1, \ldots, y_5 \). Note that the original length is \( n = 10 \). The traces are noisy copies of \( x \) and includes deletions, insertions and substitutions.

Figure 5.6 presents the process of computing the patterns of \( (y_1, y_2), (y_1, y_3), (y_1, y_4), (y_1, y_5) \). For each pair, \( y_1 \) and \( y_i \), Figure 5.6 depicts \( \mu \), which is the LCS of the sequences \( y_1 \) and \( y_i \). Then, the figure presents \( u_{y_1, \mu} \), which is the embedding sequence that the algorithms uses in order to compute the patterns. Lastly, the list of patterns of each pair is depicted in increasing order of their indices. Note that lowercase symbols present gaps and \( X \) presents the symbol \( ? \). Figure 5.7 summarizes the patterns and their frequencies. Each list includes pattern from specific index. The numbers on the left side of each list are the indices. The numbers on the right side of each pattern in a list represents the pattern’s frequency. It is not hard to observe that the longest path in the pattern path graph of this example is:

\[ \text{GT} \rightarrow \text{GTA} \rightarrow \text{TAX} \rightarrow \text{AXG} \rightarrow \text{XGT} \rightarrow \text{GTG} \rightarrow \text{TGC} \rightarrow \text{GCC} \rightarrow \text{CCtG} \rightarrow \text{CtG} \rightarrow \text{G} \]

and the algorithm output will be \( \hat{y}_1 = \text{GTAGTGCCTG} = x \).

5.4.3 The Divider BMA Algorithm

In this section we present Algorithm 10, which is a variation of the algorithms from [5] and [26]. The algorithm receives \( C \), a cluster of traces, and divides it to three different sub-clusters by their lengths. The sub-cluster \( C_n \) contains traces from \( C \) of length \( n \), \( C_{<n} \) contains traces from \( C \) with length smaller than \( n \), and \( C_{>n} \) contains traces from \( C \) with length larger than \( n \). The algorithm creates a \( 5 \times n \) matrix \( M \), which is referred by the majority matrix. The \( j \)-th column of the the 1-st, 2-nd, 3-rd and 4-th rows in the matrix \( M \) counts the occurrences in position \( j \) of the symbols A, C, G, T respectively. The \( j \)-th column of the last row holds the number of occurrences in position \( j \) of the symbol that has the largest number of occurrences out of the four symbols. The algorithm starts with \( C_n \) and updates the matrix such that \( M_{i,j} = \{|c(j) = i : c \in C_n\}| \), then it updates the 5-th row of the matrix, to have in the \( j \)-th column the maximal value out of the values of the \( j \)-th column of the first 4 rows. The majority sequence \( \hat{x} \) in index \( j \) is defined to be \( \text{arg max}\{|M_{i,j}|0 \leq i \leq 3\} \). Then, the algorithm continues with the sub-cluster \( C_{<n} \). The algorithm iterates over the traces in \( C_{<n} \), and for each trace \( y_k \) in \( C_{<n} \) the algorithm initializes a pointer \( i = 0 \). Then, the algorithm also initializes a pointer \( j = 0 \), which is used as a pointer for the majority sequence.
The lengths of the traces in this sub-cluster are smaller than $n$. Hence, they have more deletion errors rather than insertion errors. Therefore, the algorithm uses the lookahead technique as presented in [25] to find and correct deletion and substitution errors in these traces, while updating the majority sequence $\hat{x}$ if it changes. The lookahead technique in Algorithm 10 works as follows. If the $i$-th symbol of a trace $y_k$ matches the $j$-th symbol of the majority sequence $\hat{x}$, the algorithm increments both pointers by one and updates $M$ accordingly. If the $i$-th symbol of a trace does not match the $j$-th symbol of the majority sequence, but the following two symbols match the following two symbols of the majority sequence, the algorithm refers to it as a substitution error. Therefore, the algorithm increments the cell in the $j$-th column of the $y_k(i)$-th row in $M$. Then, the algorithm checks if the value in this cell is the maximal value of all values in the $j$-th column of $M$. If this condition holds, the algorithm updates the $j$-th symbol of $\hat{x}$ to be $y_k(i)$. Then, the algorithm increments the pointers $i$ and $j$ by one. In the following step, the algorithm checks if the $i$-th symbol of $y_k$ does not match the $j$-th symbol of the majority sequence $\hat{x}$, but the value in the $j$-th column of the $y(i)$-th row of $M$ is smaller than the value in the $j$-th column of the 5-th row of $M$ by two or less. If this conditions holds, the algorithm increments by one the value in the $j$-th column of the $y(i)$-th row of $M$. Then, the algorithm updates the $j$-th symbol of $\hat{x}$ to be the symbol that its row in the matrix $M$ has the maximum value in the $j$-th column. Then, the algorithm increments pointer $i$ and pointer $j$ by one. If none of the above conditions hold, and $d > 0$, the algorithm refers to the $j$-th symbol of $y_k$ as a deleted symbol. Therefore, the algorithm increments pointer $j$ by one and decrement $d$ by one. If $d \leq 0$ the algorithm increments $i$ by one, and continues to the next trace. Lastly, the algorithm uses the same lookahead technique on the sub-cluster $C_{>n}$ to detect and correct insertion errors and returns the majority sequence $\hat{x}$.

5.5 Results

In this section we present evaluation of the accuracy of Algorithm 4 and Algorithm 5 on simulated data and on data from DNA storage experiments. We also
Algorithm 10 Divider BMA

Input:
- Design Length = \(n\)
- Cluster \(C\) of \(t\) noisy copies: \(y_1, y_2, \ldots, y_t\), divided to three sub-clusters:
  - \(C_n\) - copies from \(C\) with length \(n\).
  - \(C_{<n}\) - copies from \(C\) with length smaller than \(n\).
  - \(C_{>n}\) - copies from \(C\) with length larger than \(n\).

Output:
- \(\hat{x}\) - Estimation of the original sequence

1. Define \(5 \times n\) matrix \(M\)
2. For \(i \in [0,3], j \in [0,n-1]\) set \(M_{i,j} = |\{c(j) = i : c \in C_n\}|\)
3. Set \(M_{4,j} = \max_{0 \leq i \leq 3} \{M_{i,j}\}\)
4. Set \(\hat{x}(j) = \arg\max\{M_{i,j} | 0 \leq i \leq 3\}\) (in case of tie, choose randomly).
5. Update matrix \(M\), using the copies from \(C_{<L}\).
   - For each copy \(y_k \in C_{<L}\):
     (a) Set \(i = 0\) and \(j = 0\). \(*i\) iterates over \(c_k\) and \(j\) iterate over \(\hat{x}\) \(*j\).
     (b) Set \(d = n - |y_k|\).
     (c) if \(y_k(i) \neq \hat{x}(j)\) then
       \[ M_{y_k(i),j} = M_{y_k(i),j} + 1 \]
       \[ j = j + 1, i = i + 1 \]
     else if \(y_k(i) = \hat{x}(j)\) and \(y_k(i+1) = \hat{x}(j+1)\) and \(y_k(i+2) = \hat{x}(j+2)\) then
       \[ M_{y_k(i),j} = M_{y_k(i),j} + 1 \]
       if \(M_{y_k(i),j} > M_{4,j}\) then \(x = y_k(j)\).
       \[ j = j + 1, i = i + 1 \]
     else if \(y_k(i) \neq \hat{x}(j)\) and \(M_{y_k(i),j} + 3 > M_{4,j}\) then
       \[ M_{y_k(i),j} = M_{y_k(i),j} + 1 \]
       if \(M_{y_k(i),j} > M_{4,j}\) then \(x = y_k(j)\).
       \[ j = j + 1, i = i + 1 \]
     else if \(d > 0\) then
       \[ d = d - 1, j = j + 1, i = i \]
     else
       \[ i = i + 1 \]
   end if
7. Return \(\hat{x}\).
implemented the algorithms from \cite{26} and from \cite{66}.

We compare the edit error rates and the success rates of all the algorithms. In all of the simulations, Algorithm 5 presented significantly smaller edit error rates and higher success rates. The results on the simulated data are depicted in Figure 5.8, Figure 5.9, and Figure 5.10. The results on the data from DNA storage experiments can be found in Figure 5.11.

5.5.1 Results on Simulated Data

We evaluated the accuracy of Algorithm 4 and Algorithm 5 by simulations. First, we present our interpretation of the deletion-insertion-substitution channel. In our deletion-insertion-substitution channel, the sequence is transmitted symbol-by-symbol. First, before transmitting the symbol, it checks for an insertion error before the transmitted symbol. The channel flips a coin, and in probability \( p_i \), an insertion error occurs before the transmitted symbol. If an insertion error occurs, the inserted symbol is chosen uniformly. Then, the channel checks for a deletion error, and again flips a coin, and in probability \( p_d \) the transmitted symbol is deleted. Lastly, the channel checks for a substitution error. The channel flips a coin, and in probability \( p_s \) the transmitted symbol is substituted to another symbol. The substituted symbol is chosen uniformly. In case that both deletion and substitution errors occurs in the same symbol, we refer to it as a substitution.

We simulated 100,000 clusters of sizes \( t = 6, 10, 20 \), the sequences length was \( n = 100 \). The deletion, insertion, and substitution probabilities were all identical, and ranged between 0.01 and 0.1. It means that the actual error probability of each cluster was \( 1 - (1 - p_i)(1 - p_s)(1 - p_d) \) and ranged between 0.029701 and 0.271. We reconstructed the original sequences of the clusters using Algorithm 4, Algorithm 5, and the algorithms from \cite{26} and from \cite{66}. For each algorithm we evaluated its edit error rate, the success rate, and the value of \( k_{1,\text{succ}} \). The edit error rate of Algorithm 5 was the lowest among the tested algorithms, while the algorithm from \cite{66} presented the highest edit error rates. Moreover, it can be seen that Algorithm 5 presented significantly low edit error rates value for higher values of error probabilities. In addition, Algorithm 5 also presented the lowest value of

\footnote{The parameters we used for the window size of the algorithm from \cite{26} were \( 2 \leq w \leq 4 \), and we present the results for all of them.}

\footnote{The parameters we used for the algorithm from \cite{66} were \( \ell = 5, \delta = (1 + p_s)/2, r = 2 \) and \( \gamma = 3/4 \), while for the data of the DNA storage experiments the substitution probability \( p_s \) was taken from \cite{53}.}
$k_{1_{\text{succ}}}$. For example, when the cluster size was $t = 20$ and the error probability was $p = 0.142625$, the value of $k_{1_{\text{succ}}}$ of Algorithm 5 was 2, while the other algorithms presented $k_{1_{\text{succ}}}$ values of at least 12. The results of these simulations for cluster sizes of $t = 10$ and $t = 20$ can be found in Figure 5.8, Figure 5.9, and Figure 5.10.

5.5.2 Results on Data from DNA Storage Experiments

In this section we present the results of the tested algorithms on data from DNA storage experiments [22, 27, 48]. The clustering of these data sets was made by the SOLQC tool [53]. We performed each of the tested algorithms on the data and evaluated the edit error rates. Note that, in order to reduce the runtime of Algorithm 5, we filtered clusters of size $t > 25$ to have only the first 25 traces. Also here, Algorithm 5 presented the lowest edit error rates in almost all of the tested data sets. These results are depicted in Figure 5.11.
Figure 5.1: Levenshtein error rate by the deletion probability $p$, for clusters of size 7 (left) and 10 (right). This figure presents results from Algorithm 1, the BMA algorithm [5], and the $p^t$ lower bound. Note that the LER was 0 for $p \leq 0.06$ and $p \leq 0.03$ for $t = 10$ and $t = 7$, respectively. The X-axis represents the different values of the deletion probability in the range $[0.01, 0.20]$ and the Y-axis represents the average LER of the clusters.
(a) \( k \)-error success rate by cluster size \( t \) and different values of \( k \). The X-axis represents the cluster size \( t \), the Y-axis represents the value of \( k \) for the calculation of the \( k \)-error success rate, and the Z-axis represents the \( k \)-error success rate.

(b) The values of \( k_{1 \text{ succ}} \) and \( k_{0.99 \text{ succ}} \) by cluster size \( t \). Denote that, \( k_{1 \text{ succ}} \) is the minimal Levenshtein errors, that an error correcting code must correct in order to fully reconstruct the tested clusters using Algorithm[1]. The X-axis represents the cluster size \( t \), the Y-axis represents the \( k_{j \text{ succ}} \).

Figure 5.2: \( k \)-error success rate, \( k_{1 \text{ succ}} \) and \( k_{j \text{ succ}} \) values by the cluster size \( 4 \leq t \leq 10 \). The deletion probability was 0.25.

85
Figure 5.3: $k$-mer distance demonstration for 4 traces. The original strand $x$ is of length $n = 20$. 
(a) Percents of clusters that the $r$-tuple of traces that was used by Algorithm 1 to reconstruct the original sequence $x$, was consisted of the longest 20% traces. The X-axis presents the deletion probability and the Y-axis presents the percents of the 10,000 clusters that satisfied this property.

(b) Running time in minutes of performing Algorithm 1 and Algorithm 3 on 10,000 clusters. The X-axis presents the deletion probability and the Y-axis presents the running time in minutes of performing Algorithm 1 and Algorithm 3 on 10,000 clusters of size $t = 100$.

Figure 5.4: Performance evaluation of Algorithm 1 and Algorithm 3. The simulation were for clusters of size $t = 100$, design length of $n = 200$, for each probability $p$ we simulate 10,000 clusters. In all of the simulations the original sequence was reconstructed by the algorithms, introducing a success rate of 1.

87
Original sequence $x = GTAGTGCCTG$

$y_1 = GTAGGTCCCG \quad y_2 = GTAGTCCCTG$  
$y_3 = GTAGTCCTCTG \quad y_4 = GTAGCGCCAG$  
$y_5 = GCATGCTCTG$

Figure 5.5: Example input to Algorithm 7

(a) Patterns of $y_1$ and $y_2$.  
(b) Patterns of $y_1$ and $y_5$.  
(c) Patterns of $y_1$ and $y_4$.  
(d) Patterns of $y_1$ and $y_5$.

Figure 5.6: Algorithm 7 Example - Patterns of $y_1$.

0 {GT: 3, GX: 1}  
1 {GTA: 3, GxCa: 1}  
2 {TAX: 2, TAG: 1, XcAX: 1}  
3 {AXG: 2, AXG: 1, AXX: 1}  
4 {XGT: 2, XXX: 1, XXT: 1}  
5 {GTG: 1, GTX: 1, XcG: 1, XTG: 1}  
6 {TGC: 2, TXC: 1, XcGC: 1}  
7 {GCC: 2, XCC: 1, GGcG: 1}  
8 {CGC: 2, Cca: 1, CcTcG: 1}  
9 {CtG: 3, CcG: 1}  
10 {G: 4}

Figure 5.7: Algorithm 7 Example - Patterns of $y_1$, their indices and their frequencies. The numbers on the left side of each list are the indices. The numbers on the right side of each pattern in a list represents the pattern’s frequency.
(a) Edit error rate by the error probability for $t = 10$. The $X$-axis represents the error probability of the simulated clusters and the $Y$-axis represents the edit error rate.

(b) Edit error rate by the error probability for $t = 20$. The $X$-axis represents the error probability of the simulated clusters and the $Y$-axis represents the edit error rate.

Figure 5.8: Edit error rate by the error probabilities for the cluster sizes $t = 10$ and $t = 20$. The length of the original sequence was $n = 100$ and the error probabilities ranges between 0.029701 and 0.271.
(a) Success rate by the error probability for $t = 10$. The X-axis represents the error probability of the simulated clusters and the Y-axis represents the success rate.

(b) Success rate by the error probability for $t = 20$. The X-axis represents the error probability of the simulated clusters and the Y-axis represents the success rate.

Figure 5.9: Success rate by the error probabilities for the cluster sizes $t = 10$ and $t = 20$. The length of the original sequence was $n = 100$ and the error probabilities ranges between 0.029701 and 0.271.

90
(a) $k_{1_{\text{succ}}}$ values by the error probability for $t = 10$. The X-axis represents the error probability and the Y-axis represents the value of $k_{1_{\text{succ}}}$.

(b) $k_{1_{\text{succ}}}$ values by the error probability for $t = 20$. The X-axis represents the error probability and the Y-axis represents the value of $k_{1_{\text{succ}}}$.

Figure 5.10: $k_{1_{\text{succ}}}$ values by the error probabilities for the cluster sizes $t = 10$ and $t = 20$. The length of the original sequence was $n = 100$ and the error probabilities ranges between 0.029701 and 0.271.
Figure 5.11: Edit error rate by the reconstruction algorithm, for data from DNA storage experiments [22, 27, 48].
Chapter 6

Conclusions and Future Work

6.1 Summary
We presented in this work several new algorithms for the deletion DNA reconstruction problem and for the DNA reconstruction problem. While most of the previously published algorithms use a symbol-wise majority approaches, our algorithms look globally on the entire sequence of the traces, and use the LCS or SCS of a given set of traces. Our algorithms were designed to support DNA storage systems and to reduce the edit error rate of the reconstructed sequences. According to our tests on simulated data and on data from DNA storage experiments, we found out that our algorithms significantly reduced the error rates compared to the previously published algorithms. Moreover, our algorithms performed even better when the error probabilities were high, while using less traces than the other algorithms.

6.2 Future Work
Even though our algorithms improved previous results, there are still several challenges that need to be addressed in order to fully solve the DNA reconstruction problem. We list the following directions for future work.

1. Design error correcting codes and coding schemes for DNA-storage systems.
2. Design DNA-storage experiments to evaluate other aspects of our algorithms.
3. The presented algorithms were designed to work with different cluster sizes. However, as presented in Section 5.3.3 in cases where the cluster is of large size, some of the traces can be filtered out to reduce the complexity and the computation time of the reconstruction process. Hence, we think that future work should focus on defining and evaluating filtering criteria for large clusters.
Acknowledgments

The authors thank Alexander Yucovich for his ideas and algorithms for Chapters 4 and 5 and its help with the simulations. They also thank Guy Shapira for his great contribution.

They also thank Prof. Zohar Yakhini for its kind and helpful guidance in Chapter 3 and for Yoav Orlev, Roy Shafir and Leon Anavy for co-writing the SOLQC tool.

The authors thank Matika Lidgi for her help in the Divider BMA algorithm. They also thank her, along with Danit Goldberg, Amir Biran, Batel Carmona, Rotem Samuel, Guy Shapira, Idan Raz, Ron Yizhak and Dafna Regev for they contribution to this work.

The authors thank Prof. Gala Yadgar for sharing her servers for the simulations in this research.

The authors thank Lee Organick, Hossein Yazdi, Karin Strauss, Yaniv Erlich, Roee Amit, Sarah Goldberg and Cyrus Rashtchian for valuable discussion.
Bibliography


[50] W. Pan, M. Byrne-Steele, C. Wang, S. Lu, S. Clemmons, R. J. Zahorchak, and J. Han, "DNA polymerase preference determines primer efficiency", *BMC Biotechnology*, vol. 14, no.1, pp. 10, 2014.


אלגוריתמי חיזור עבורי

מערבוט אוחסן מבוססות דנא

ועמר צבריא
אלגוריתמי חיזור עבורה
מערבות אתחוליות מבוססות דגא

חנור על מחקר

לשם מילוי חלקי של הדרישותלקבלת תואר
morgan lurkingזמצוי במדעים המחשב

עומר צבי

הנהטנשוב טכנולוגיה – מרכז טכנולוגיה לישראל
22.06.20 חיפה

ל בסיון תשיש
הכרת תודה

ברצוני להודות לمنتה שליל, פרופסור י TripAdvisor, עלי
הממקצעיות, המסרירת, הת�יכתה והחדירה לאורח הדרכ. את
מַהוּה עֵבֹרֵי דְוֹמְה בֵּעֲבַדְתּךְ המקצועיות ביד어야כתיי
-אָסִיָּאִים

אני מודה ל_spell governments על התמיכתה כספית והדרכה ב nuesה
,International Symposium on Information Theory יז
2020

עבודיה הוצגה בניו ה whore היותר

ב

Technion - Computer Science Department - M.Sc. Thesis MSC-2020-25 - 2020
הצタイプ

 TRACE RECONSTRUCTION PROBLEM

נתונות מיקורי $x$ ומיקורי $t$ המ璨ים סיביות בканצל $t$ של מיקוד $t$. במקורי $t$ יש $m$ סיביות שמתאימות עם שאר מיקוד $t$. בערך המיקוד $t_d$ במקורי $t$ כך שבו כל מקוד $t_d$ נגזר משאר מיקוד $t$ שאף אחד מהמקודים $t$ שמקודים $t_d$уютו כדי ש$t_d$ ו$t$ י買い תפסנים זה אחר זה.

$\psi(x)$ מייצגת התווים אפשריים של מקוד $t$, כאשר $\mathcal{P}$ פלט מקוד אפליקציה $\mathcal{F}$ יהיה מקוד $t$. $\Lambda$ מייצג את מקוד $x$. $\mathcal{S}$ מייצג את מקוד $t$.

$\mathcal{P}$ מייצג את מקוד אפליקציה $\mathcal{F}$,$\Lambda$ מייצג את מקוד $x$ ו$\mathcal{S}$ מייצג את מקוד $t$.

$\mathcal{S}$ מייצג את מקוד אפליקציה $\mathcal{F}$,$\Lambda$ מייצג את מקוד $x$ ו$\mathcal{S}$ מייצג את מקוד $t$.

$\mathcal{S}$ מייצג את מקוד אפליקציה $\mathcal{F}$,$\Lambda$ מייצג את מקוד $x$ ו$\mathcal{S}$ מייצג את מקוד $t$.

$\mathcal{S}$ מייצג את מקוד אפליקציה $\mathcal{F}$,$\Lambda$ מייצג את מקוד $x$ ו$\mathcal{S}$ מייצג את מקוד $t$.

$\mathcal{S}$ מייצג את מקוד אפליקציה $\mathcal{F}$,$\Lambda$ מייצג את מקוד $x$ ו$\mathcal{S}$ מייצג את מקוד $t$.

$\mathcal{S}$ מייצג את מקוד אפליקציה $\mathcal{F}$,$\Lambda$ מייצג את מקוד $x$ ו$\mathcal{S}$ מייצג את מקוד $t$.

$\mathcal{S}$ מייצג את מקוד אפליקציה $\mathcal{F}$,$\Lambda$ מייצג את מקוד $x$ ו$\mathcal{S}$ מייצג את מקוד $t$.

$\mathcal{S}$ מייצג את מקוד אפליקציה $\mathcal{F}$,$\Lambda$ מייצג את Макוד $x$ ו$\mathcal{S}$ מייצג את מקוד $t$.

$\mathcal{S}$ מייצג את מקוד אפליקציה $\mathcal{F}$,$\Lambda$ מייצג את מקוד $x$ ו$\mathcal{S}$ מייצג את מקוד $t$.

$\mathcal{S}$ מייצג את מקוד אפליקציה $\mathcal{F}$,$\Lambda$ מייצג את מקוד $x$ ו$\mathcal{S}$ מייצג את מקוד $t$.

$\mathcal{S}$ מייצג את מקוד אפליקציה $\mathcal{F}$,$\Lambda$ מייצג את מקוד $x$ ו$\mathcal{S}$ מייצג את מקוד $t$.

$\mathcal{S}$ מייצג את מקוד אפליקציה $\mathcal{F}$,$\Lambda$ מייצג את מקוד $x$ ו$\mathcal{S}$ מייצג את מקוד $t$.

$\mathcal{S}$ מייצג את מקוד אפליקציה $\mathcal{F}$,$\Lambda$ מייצג את מקוד $x$ ו$\mathcal{S}$ מייצג את מקוד $t$.

$\mathcal{S}$ מייצג את מקוד אפליקציה $\mathcal{F}$,$\Lambda$ מייצג את מקוד $x$ ו$\mathcal{S}$ מייצג את מקוד $t$.

$\mathcal{S}$ מייצג את מקוד אפליקציה $\mathcal{F}$,$\Lambda$ מייצג את מקוד $x$ ו$\mathcal{S}$ מייצג את מקוד $t$.

$\mathcal{S}$ מייצג את מקוד אפליקציה $\mathcal{F}$,$\Lambda$ מייצג את מקוד $x$ ו$\mathcal{S}$ מייצג את מקוד $t$.

$\mathcal{S}$ מייצג את מקוד אפליקציה $\mathcal{F}$,$\Lambda$ מייצג את מקוד $x$ ו$\mathcal{S}$ מייצג את מקוד $t$.

$\mathcal{S}$ מייצג את מקוד אפליקציה $\mathcal{F}$,$\Lambda$ מייצג את מקוד $x$ ו$\mathcal{S}$ מייצג את מקוד $t$.

$\mathcal{S}$ מייצג את מקוד אפליקציה $\mathcal{F}$,$\Lambda$ מייצג את מקוד $x$ ו$\mathcal{S}$ מייצג את מקוד $t$.

$\mathcal{S}$ מייצג את מקוד אפליקציה $\mathcal{F}$,$\Lambda$ מייצג את מקוד $x$ ו$\mathcal{S}$ מייצג את מקוד $t$.

$\mathcal{S}$ מייצג את Макוד אפליקציה $\mathcal{F}$,$\Lambda$ מייצג את מקוד $x$ ו$\mathcal{S}$ מייצג את מקוד $t$.

$\mathcal{S}$ מייצג את מקוד אפליקציה $\mathcal{F}$,$\Lambda$ מייצג את מקוד $x$ ו$\mathcal{S}$ מייצג את מקוד $t$.

$\mathcal{S}$ מייצג את מקוד אפליקציה $\mathcal{F}$,$\Lambda$ מייצג את מקוד $x$ ו$\mathcal{S}$ מייצג את מקוד $t$.

$\mathcal{S}$ מייצג את מקוד אפליקציה $\mathcal{F}$,$\Lambda$ מייצג את מקוד $x$ ו$\mathcal{S}$ מייצג את מקוד $t$.

$\mathcal{S}$ מייצג את מקוד אפליקציה $\mathcal{F}$,$\Lambda$ מייצג את מקוד $x$ ו$\mathcal{S}$ מייצג את מקוד $t$.

$\mathcal{S}$ מייצג את מקוד אפליקציה $\mathcal{F}$,$\Lambda$ מייצג את מקוד $x$ ו$\mathcal{S}$ מייצג את מקוד $t$.

$\mathcal{S}$ מייצג את מקוד אפליקציה $\mathcal{F}$,$\Lambda$ מייצג את מקוד $x$ ו$\mathcal{S}$ מייצג את מקוד $t$.
The eligibility criterion for the algorithms implemented by the thesis is the following: the algorithms were designed to find the shortest common supersequence (Shortest Common Supersequence) and the longest common subsequence (Longest Common Subsequence) of two given sequences.

The algorithms are as follows:

- **Shortest Common Supersequence (SCS):**
  - The algorithm finds the shortest sequence that contains both input sequences as subsequences.
  - Time complexity: $O(\min(m,n) \times m \times n)$.

- **Longest Common Subsequence (LCS):**
  - The algorithm finds the longest sequence that is a subsequence of both input sequences.
  - Time complexity: $O(m \times n)$.

The algorithms were implemented in C++ and tested on various datasets. The results showed that the algorithms are efficient and effective in finding the required sequences.

The implementation was done using the Boost C++ Libraries for linked lists and dynamic programming.

Further research could include improving the algorithms for larger datasets or implementing other sequence comparison algorithms.