USING DNA FOR DATA STORAGE: ENCODING AND DECODING ALGORITHM DEVELOPMENT

by

Kelsey Suyehira

A thesis

submitted in partial fulfillment

of the requirements for the degree of

Master of Science in Computer Science

Boise State University

September 2018

BOISE STATE UNIVERSITY GRADUATE COLLEGE

DEFENSE COMMITTEE AND FINAL READING APPROVALS

of the thesis submitted by

Kelsey Suyehira

Thesis Title: Using DNA for Data Storage: Encoding and Decoding Algorithm Development

Date of Final Oral Examination: 7 September 2018

The following individuals read and discussed the thesis submitted by student Kelsey Suyehira, and they evaluated the presentation and response to questions during the final oral examination. They found that the student passed the final oral examination.

Tim Andersen, Ph.D.	Chair, Supervisory Committee
Reza Zadegan, Ph.D.	Member, Supervisory Committee
Catherine Olschanowsky, Ph.D.	Member, Supervisory Committee

The final reading approval of the thesis was granted by Tim Andersen, Ph.D., Chair of the Supervisory Committee. The thesis was approved for the Graduate College by Tammi Vacha-Haase, Ph.D., Dean of the Graduate College.

Dedicated to my Mom, Connie Suyehira.

ACKNOWLEDGMENTS

The author wishes to thank all who provided insight and helped her with the completion of this thesis. This work would not have been possible without the support of many faculty and staff from within and outside of the computer science department.

ABSTRACT

The recent explosion of digital data has created an increasing need for improved data storage architectures with the ability to store large amounts of data over extensive periods of time. DNA as a data storage solution shows promise with a thousand times greater increase in information density and information retention times ranging from hundreds to thousands of years. This thesis explores the challenges and potential approaches in developing an encoding and decoding algorithm for use in a DNA data storage architecture. When encoding binary data into sequences representing DNA strands, the algorithms should account for biological constraints representing the idiosyncrasies of working with a molecular substance. We present REDNAM (Robust Encoding and Decoding of Nucleic Acid Memory). REDNAM includes a novel mapping scheme and translation stage which converts hexadecimal data to codons while accounting for four constraints; removing start codons, avoiding repeating nucleotides, excluding longer repeating sequences, and maintaining close to 50% GC content. We have integrated this mapping scheme into the Fountain Codes algorithm in an implementation that balances information density with error correction and parity data. Preliminary results show that our implementation can successfully recover the original dataset after artificial insertion, deletion and mutation errors have randomly perturbed the encoded information. We also achieved a speed up of two times for encoding and 435 times for decoding compared to another Fountain Codes implementation.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

DNA – Deoxyribonucleic Acid

RNA – Ribonucleic Acid

NAM – Nucleic Acid Memory

REDNAM – Robust Encoding and Decoding of Nucleic Acid Memory

nt – Nucleotide

CRC – Cyclic Redundancy Check

PRNG – Pseudo Random Number Generator

LFSR – Linear-Feedback Shift Register

XOR – Exclusive-Or

FIFO – First-In-First-Out

LIST OF SYMBOLS

K	The number of input segments
S	The expected number of single-segment packets
δ	Distribution parameter for robust Soliton distribution
c	Distribution parameter for robust Soliton distribution

CHAPTER 1

INTRODUCTION

Technological advances in personal computing and computer enabled devices has caused digital technology to become a consistent part of people's everyday lives. A growing number of us interact with web and mobile applications that track our uses, multiple times a day. There is an increasing utilization of technology in numerous industries. Data collection is now ubiquitous. Personal and environmental data are constantly collected. This includes: personal health data, public government records, Facebook likes, and weather sensor information. Industries are reaching a point where they are trying to keep this ever expanding amount of information valid and accessible over extended periods of time. Consistent long-term data storage at this volume requires innovation beyond traditional data storage techniques. This thesis explores the challenges and possible approaches to storing large data sets using synthetic DNA.

By the late 1990's, not all researchers anticipated the enormity of the impending data explosion. To show the recent growth, in May of 2013, one researcher estimated that 90% of the data in the world was generated in just the previous two years [33]. However, in a statement recorded in 1997, after estimating that all of the world's data amounted to about 12 Exabytes [24], another researcher concluded that "... storage media will outrun our ability to create things to put on them ..." [23]. They did not

foresee this substantial growth of data collection and the need to store large amounts of it for such long periods of time. Many data storage technologies were only made to last ten to twenty years with some technologies now becoming obsolete.

It is now clear that the rate of data collection and the variety of data being collected will continue to increase. Novel technologies and technological advances will continue to expand to try and fill this need. Some types of data need to be accessed often, but information such as scientific, financial, government, historical, genealogical, and genetic records can be stored away for safekeeping for hundreds of years. In order to combat the excessive use of resources to store huge amounts of data, especially over long periods of time, researchers are seeking alternative storage techniques.

A system in which data could be stored for hundreds or even thousands of years is highly desirable. Deoxyribonucleic acid (DNA) as a data storage material is a promising concept. This organic material is composed of four unique chemical components and is used by organisms to store genetic information. Developing an efficient algorithm to convert arbitrary data to stable strands of quaternary DNA bases, and then back again, can open the door to a new kind of long-term data storage solution.

1.1 DNA as a Data Storage Device

DNA is a durable material that when stored under the right conditions can be stable for hundreds of years [37]. Potentially, DNA can last for longer periods of time, and complete genomes have even been sequenced from a ∼50,000 year old Neanderthal [37]. As a data storage method, DNA can be stored in water or dry air, for long

durations. Under degradation tests completed at 10 degrees Celsius, it was calculated that DNA has retention values ranging from $2x10^4$ years in water to $2x10^7$ years in air. Those researchers stated that the effect of hydrolysis is greater than temperature when it comes to the degradation of DNA [37]. This shows that under the right conditions, centuries of historical records could potentially be stored for thousands of years.

Another advantage of DNA as a storage material is its high data density. Molecules are nanoscale in size, therefore, the physical space needed to store DNA is quite minuscule. Researchers estimate that today's global storage needs, about 10^{22} bits, could be stored within a $10x10x10cm^3$ box [37], about the size of a square tissue box. These estimates assume ideal encoding schemes, in which there is a direct translation from bits to nucleotides. However, incorporating other necessary information, such as duplication or parity for error resilience, can cut down on the information density. Yet, utilizing DNA as a storage material can still provide large improvements in information density over current technologies. For example, a hard disk can store about 10^{13} bits per cm^3, while DNA can potentially reach a volumetric density of about 10^{19} bits per cm^3 [37], or several orders of magnitude more dense.

Composing a DNA storage solution involves constructing an algorithm for encoding and decoding data from binary to quaternary. An intelligent way to convert 0's and 1's into the four molecular bases or nucleotides - adenine (A), thymine (T), cytosine (C), and guanine (G) - that make up DNA, one that takes into account the idiosyncrasies of DNA, is needed. Once a mapping scheme is developed, digital data is converted to sequences of the nucleotide characters. Then, using those sequences, DNA synthesis technologies are used to piece molecules together to create the physical strands of DNA. The physical, single-stranded DNA is then stored in a cool, dry

place for an indefinite amount of time. Once the DNA is ready to be decoded, DNA sequencing technologies are used to read back the strands of nucleotide sequences. Finally, the sequences are passed back through the decoding algorithm to recover the original data. An overview of this process is shown in Figure 1.1. The right half of this diagram pictures the physical synthesis and sequence stages while the left half of the diagram represents the conversion of binary data to nucleotides that can be done programmatically. An understanding of the right half of this diagram is needed in order to better develop the left half of the diagram, which is the focus of this thesis.

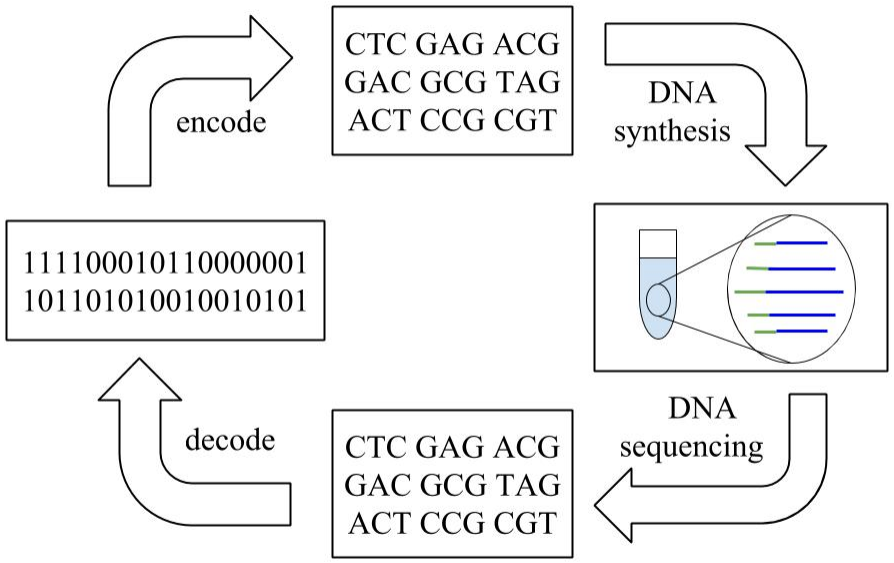


Figure 1.1: Process Overview

1.2 Thesis Statement

We present a novel DNA encoding and decoding algorithm that can be used to encode and store arbitrary information in DNA molecules. Unlike other encoding schemes, our algorithm accounts for biological factors and other DNA-specific issues that can compromise the ability to encode and recover data accurately from DNA, and also addresses environmental concerns. Additionally, the algorithm utilizes Fountain Codes, optimizing for both information density and error recovery, while maintaining feasible computational bounds.

1.3 Contributions

The main contributions presented in this thesis are summarized as follows:

1. A novel mapping scheme for converting hexadecimal data to codons that accounts for biological constraints and can be incorporated into multiple different encoding and decoding algorithms.

2. Analysis of the ability of the Fountain Codes algorithm to accurately decode messages depending on different parameters of the Luby Transform.

3. An implementation of the encoding and decoding algorithms that convert digital data to sequences of nucleotides and back again. The program includes optimizations made to the Fountain Codes algorithm as well as the integration of our new mapping scheme. The result is an application that can more accurately recover data than previous implementations with a running time that is fast enough to encode and decode large data sets.

1.4 Thesis Organization

This thesis details research on developing algorithms for encoding data to, and decoding data from DNA sequences in six chapters. This chapter includes an overview of using DNA for data storage. It also includes the thesis statement and a list of the main contributions.

Chapter two contains background information which starts with an overview of the current data storage techniques in production and includes a comparison of those techniques with a DNA data storage system. The next part of chapter two covers constraints and considerations that need to be accounted for in creating a DNA data storage application. Finally, chapter two ends with some background information on information theory and error correction codes, specifically Reed Solomon codes.

Chapter three is the related works section. This section includes a survey of seven other proposed encoding and decoding algorithms. Each algorithm is explained in detail and follows with a discussion of highlights and possible areas for improvement.

The contributions of this thesis are reported in chapter four. The first part of this chapter describes a new mapping scheme for converting binary to nucleotides. This mapping scheme accounts for multiple biological constraints. The next section of chapter four includes analyses of the Fountain Codes algorithm and the implementation used in Erlich and Zielinski's algorithm called DNA Fountain. The last part of chapter four details the integration of the new mapping scheme into the Fountain Codes algorithm and other improvements that were made to the algorithm design.

Chapter five describes the results of this research. This section contains a description of validation tests that were run and the corresponding results. We have shown that our algorithm can recover from high levels of insertion, deletion, and mutation

errors. We also found that Erlich and Zileinksi's proposed algorithm is unable to successfully decode an encoded dataset in some cases, and when compared to their implementation, we produced a speed up of 2 times for encoding and 435 times for decoding, in terms of running time.

The last chapter of this paper is the conclusion and includes possible areas for continued work. This paper ends with an appendix that includes documentation for the code implementation.

CHAPTER 2

BACKGROUND

The multi-disciplinary nature of this thesis project incorporates knowledge from other areas of research including engineering, materials science and mathematics. This chapter presents background information on those topics in order to better understand the research discussed in this thesis. The first section of this chapter focuses on engineering, where researchers and developers have been working to increase the storage capacity of current data storage techniques and to extend the lifespan of their products. In the second section, information from the fields of materials science, biology, and chemistry helps uncover the intricacies involved in using DNA for data storage. Molecular reactions present their own set of challenges and constraints to understand for this application. Finally, mathematicians working in the field of information theory have researched the theoretical boundaries for the ability to recover information in a sequence given potential errors. Pulling from research in these areas, this chapter covers three topics including current data storage solutions, molecular constraints, and information theory.

2.1 Current Data Storage Landscape

The current most popular data storage techniques include magnetic tapes, magnetic disks, optical devices (such as CDs, DVDs, and Blu Ray) and flash memory, including

solid state drives. The following section describes each of these technologies in detail as well as some possible advantages or disadvantages of using each technique.

Magnetic tape storage includes devices that have a long thin plastic film with a magnetic coding that is altered with magnetic impressions to represent data. Magnetic tape was originally developed for audio storage, so the data access pattern is sequential and random access requires rewinding the tape. While some data storage centers are moving away from magnetic tapes in favor of magnetic disks, the low cost per bit of magnetic tape keeps it as a viable data storage alternative. Magnetic tape also consumes much less energy than magnetic disks which is an important consideration for data centers [18]. Recently, research has been conducted into increasing the storage capacity of magnetic tape and Sony claims to have developed a magnetic tape technology with an areal density of 148 gigabits per square inch [3]. However, most magnetic tape systems are only guaranteed to retain data up to 30 years [2].

Magnetic disks, such as hard disk drives, store magnetic data on rotating disks. A magnetic head is used to read and write data. The data can be obtained in a random-access manner but the system favors sequential access. Most recently, hard disks have reached a capacity of 12 terabytes [7] and can retain data up to 5 years [15]. Hard disk drives are common as secondary storage but are starting to reach their limits in terms of growth and improvements. It is almost impossible to improve upon rotation speeds and improvements in random access times lag behind the capacity growth [18].

Optical devices, such as CD's, DVD's and Blu-ray discs, are flat, round discs that record binary information by changing photo-physical forms on the surface. Information is accessed from the disks by emitting a light beam against the surface and

sensing the reflection. Optical disks are common for distributing information because of their low manufacturing cost. However, optical disks are not widely used for data storage centers because they lag behind magnetic tape in terms of recording density [18]. Although, Sony and Panasonic have recently partnered to create an optical disk archival storage system. The companies have created a roadmap to develop a system with a recording capacity of 300 gigabytes per disk in the first generation to 1 terabyte in the third generation [5]. They claim that their system will have an estimated lifespan of 50 years.

Both magnetic disk devices and optical devices are in competition with and starting to be replaced by flash memory such as flash drives and solid-state drives. Flash memory can be electronically programmed by putting or removing an electron to or from a floating gate. Information can then be read by measuring the voltage from the floating gate. Some advantages of flash memory over magnetic disks include dramatically reduced latencies in terms of data access times, and flash memory does not have mechanical limitations. However, the cost per bit for flash memory is significantly higher than magnetic disks. Yet, some manufacturers of flash memory promise retention times of up to 20 years [6]. In 2016, Samsung announced a 32 terabyte solid state drive, which they hope to increase up to 100 terabytes by 2020 [12].

DNA as a storage solution shows improvement over these current techniques in two main areas. The first is the retention times of the data. DNA can potentially last hundreds to thousands of years under the right conditions, significantly outlasting current data retention times. Secondly, the volumetric density of DNA is one thousand times greater than flash memory. Potentially, data could be stored in much smaller spaces instead of large data storage centers. Table 2.1 shows a comparison of

DNA for data storage with the four current techniques previously described [37].

Table 2.1: Data Storage Comparison

	Magnetic Tape	Magnetic Disk (Hard Disk)	Optical Drives (Blu Ray)	Flash Memory (SSD)	Cellular DNA
Retention Times	30 years	5 years	50 years	20 years	>100 years
Areal Density	10^9bit/cm^2	10^2bit/cm^2	10^{11}bit/cm^2	10^{10}bit/cm^2	unavailable
Volumetric Density	10^{12}bit/cm^3	10^{13}bit/cm^3	10^{12}bit/cm^3	10^{16}bit/cm^3	10^{19}bit/cm^3
Max Capacity	185 TB	12 TB	300 GB	32 TB	unavailable
Read/Write Latency	unavailable*	3-5ms/bit	unavailable*	100us/bit	<100us/bit
ON Power	unavailable	0.04W/GB	unavailable	0.001-0.004 W/GB	<10-10W/GB

* Values may vary depending on streaming data or a singular access, as well as the location of an access and whether the system needs to be rewound.

Since the idea of using DNA for data storage is relatively new, there are a few disadvantages when compared to current architectures. One is the high cost of producing the DNA strands themselves and reading them back to retrieve the data. Currently, the estimated cost of synthesizing 1 megabyte of data into DNA is over $12,000 and $220 for sequencing the DNA back again [28]. This seems high, but the cost of DNA sequencing is already one three-millionth what it was 10 years ago [28].

Along with the high cost, it is also slow to synthesize and sequence the DNA strands. This translates to slow read and write times. These slow access times are one of the reasons why DNA data storage would be best for long-term data storage in which the information did not need to be accessed frequently. Although, the

synthesis and sequencing technologies are currently being improved for use in multiple industries. Improvements made for these processes could further lower the cost and decrease read and write times in the future. For now, researchers should focus on optimizing computational running times for the encoding and decoding algorithms.

It is clear that the sheer amount of data generated and stored each day can cause problems that need to be addressed. One of those is the large amounts of energy that current data storage centers require to load data for storage and retrieve the information when needed. According to the Natural Resources Defense Council, in 2013, U.S. data centers consumed an estimated 91 billion kilowatt-hours of electricity, which is equivalent to the annual output of 34 large coal-fired power plants [16]. Since these current storage techniques do not last, after around a decade or two, failing machines need to be replaced and the data copied over to new machines. In 2015, IBM estimated that 2.5 quintillion bytes of data are generated each day [4]. Some researchers suggest that by 2040, world storage needs will be about $2\text{x}10^{24}$ bits of information [37]. With more and more data being produced every day, industries could perceivably end up in a situation in which the rate at which data needs to be refreshed is outpaced by the time it takes to copy over the data. While research is taking place on other novel data storage architectures, this thesis focuses on developing the encoding and decoding algorithms needed in order to store data in DNA.

2.2 Constraints and Considerations

In developing a reliable encoding and decoding algorithm, the processes of physically synthesizing and sequencing the DNA must be considered. For example, a straight

mapping between binary and quaternary does not take into consideration many of the constraints that can complicate the synthesis and sequencing processes. While DNA is a durable material, it is also a natural one that has intricate enzymatic properties. Not all molecular interactions can be known and accounted for. Errors may be introduced into the DNA strands if molecular interactions interfere with the ability to synthesize and sequence the DNA correctly. Current synthesis and sequencing technologies are being improved upon, but are not always precise. However, following general rules allows for an encoding algorithm to generate DNA sequences that are more likely to produce accurate strands and readings by the synthesis and sequencing processes.

Three different types of errors can occur on the nucleotide level during these processes, especially if DNA strands are likely to have molecular reactions. These types of errors are insertions, deletions and mutations. A mutations occurs when one nucleotide is transformed into another. In consideration of these processes, a robust encoding scheme would construct strands that are less likely to cause reactions, and in turn, more likely to be synthesized and sequenced correctly. The following constraints should be considered by the encoding and decoding algorithm.

Producing sequences with repeating nucleotides should be avoided when developing an encoding algorithm. In this category, the simplest constraint to consider is avoiding consecutive bases. Sequences containing more than 3 or 4 consecutive bases in a row, also known as homopolymers, can be difficult for the synthesis and sequencing processes to handle.

The same is true for avoiding longer sequences of repeating nucleotides. If one long sequence of nucleotides appears in a strand, then that same exact sequence should not appear elsewhere. For example, if a strand contained the sequence GGACTTCGAAT, then that same sequence should not appear elsewhere in the same strand. DNA

naturally forms a double helix structure in which A pairs with T and C pairs with G on opposite strands. These molecular forces can cause a strand with repeat sequences to sometimes fold back on top of itself and create bonds. This formation is known as a hairpin structure. Because of the geometry of the molecules, a hairpin structure can make sequencing the DNA much more difficult and the information that is read back may not be accurate.

The chance of forming hairpin structures can be lessened by avoiding repeating sequences as well as palindromic sequences. Palindromic sequences are two sequences which are made up of complementary nucleotides that are in reverse order of each other. For example ATTCAGGC and GCCTGAAT are palindromic sequences.

Finally, the encoding algorithm should attempt to construct DNA strands that have close to 50% GC content. This means that for a given strand, about half of the sequence should be made up of the nucleotides C and G, and the other half should include the nucleotides A and T. Strands that have close to 50% GC content are more stable [34] and therefore, more likely to be synthesized and sequenced accurately.

Unfortunately, even though the possibility of introducing errors is reduced when an encoding algorithm has evaluated all of these constraints, accounting for these constraints does not guarantee perfect synthesis and sequencing of the DNA strands. The technology is still being improved upon, and the ability to perfectly construct and read back DNA strands without error, every time, is not yet possible. Thus, an important element in the development of an encoding/decoding algorithm is to include a way to recover insertion, deletion, and mutation errors with an additional error correction code. The simplest way to do this is with high amounts of duplication. The algorithm can then employ majority voting on all of the sequenced DNA strands. Although, too much duplication is inefficient and can cut down on the information

density. This presents a trade off in terms of the amount of error correction included in contrast with a high information density. The information density should be maximized while allowing for the complete recovery of data without error when developing an algorithm for encoding and decoding DNA.

2.3 Information Theory

The upper bounds on the amount of information that can be successfully recovered without error in a given DNA data storage system can be better understood by drawing from research within the field of information theory. This field was essentially born in 1948 with the publication of Claude E. Shannon's paper titled, "A Mathematical Theory of Communication." In this seminal paper, Shannon presented a mathematical way to quantify information. This metric can then be used to determine the minimum amount of symbols needed for the error-free representation of a given message [36]. In sending a message, the channel capacity for a discrete memoryless channel is defined as

$$C = \max_{P_X} I(X;Y)$$

where X is a given transmitted sequence, Y is the corresponding received sequence and I(X;Y) is the mutual information of X and Y [26].

Erlich and Zielinski calculated the channel capacity for their DNA data storage model in the supplementary information section of their publication. Given a mapping from two bits to one nucleotide would produce a theoretical maximum of 2 bits/nt. However, they also account for two specific biological constraints including repeats of more than three nucleotides in a row and between 45 to 55% GC content. Accounting for these constraints, Erlich and Zielinski calculated a channel capacity of 1.98 bits/nt

[17]. They then go a step further and consider the fact that some nucleotides in the sequence are needed for an index value and there may be a dropout rate of 0.5%. Including these constraints, the final calculation results in a potential capacity of 1.83 bits/nt [17].

Besides the dropout rate determined by the sequencing coverage, other errors may be introduced into the channel, as was discussed in Section 2.2. Delving further into the field of information theory can help determine an optimal amount of parity symbols necessary for the detection and correction of these errors. The next section focuses on this subfield of information theory known as channel coding which includes the study and design of error correction codes.

2.3.1 Channel Coding

Studying channel coding within the field of information theory can provide insights into recovering from errors. In a channel coding system, information is transmitted from a sender to a receiver over a noisy channel. Because of noise in the channel, information sent from the sender may be lost and the receiver is unable to collect all of the original information. The sender can also add parity information to be sent across the channel in order for the receiver to have a better chance at recovering the information, even in the event that some of it is lost. This parity information comes in the form of an error correction code. Error correction codes have been extensively studied in order to develop efficient codes that can maximize the probability of recovering information while minimizing the amount of parity information that has to be sent.

In the situation of using DNA for data storage, the processes of synthesizing, storing and sequencing the DNA can be thought of as the channel. During these

stages, errors can be introduced which may cause loss of information. Including an error correction code can increase the chance of accurately recovering the data. Some error correction codes exhibit trade offs in terms of the amount of parity information included and the probability of fully recovering errors. Certain error correction codes would be better for certain situations depending on the parameters that a system would need to optimize.

There are a few ways in which a DNA data storage channel differs from a common communication channel. Because of these differences, not all error correction codes can be directly used for DNA data storage system. One difference is that the DNA data storage channel is a one-way communication system. An advantage of a two-way communication system is the possibility for the receiver to communicate back to the sender whether or not all of the information was recovered, or if more information is needed. This is not possible in a DNA data storage system. All of the information is encoded at once and stored away. The DNA will not be decoded until potentially many years later and only the recovered sequences can be used to decode the original data.

Another difference between a common communication channel and a DNA data storage channel is the type of errors that may occur. In a typical communication channel, only deletion errors are expected. These deletions may affect single bits or entire packages of information, but the probability of each deletion is modeled to be independent and identically distributed. On the other hand, for DNA, insertion and mutation errors may also occur. Single insertion, deletion and mutation errors are possible as well as consecutive groups of errors. These various types of errors may need to be handled differently.

2.3.2 Reed Solomon Codes

One group of error correction codes that are used for multiple information technologies and have been used for DNA data storage algorithms [20, 9, 17] are Reed-Solomon codes [31]. Variants of Reed Solomon codes are used on CD's, DVD's as well as satellite links and other communication systems [21]. Grass et. al., Blawat et. al., and Erlich et. al. have all successfully employed Reed Solomon codes during lab tests of their algorithms [20, 9, 17]. Not only are these codes somewhat simple to understand, compared to other error correction codes, Reed Solomon codes can detect and correct bit flips, which translates to mutation errors. Another useful feature of these codes is that the length can be adjusted to the amount of detection and correction needed, based on prior calculations. A basic example of a Reed Solomon coding application is shown below so that the reader may gain a better understanding of how the codes work. This example is based off of the Reed Solomon tutorial by Westall, et. al. [35].

Reed Solomon codes employ algebraic properties in order to operate. One of the main concepts used by this set of codes are Galois Fields, denoted GF(p^m) which are a type of field. Fields have the following mathematical definition [36].

Let F be a set of objects on which two operations + and $\cdot$ are defined. F is said to be a **field** if and only if

1. F forms a commutative group under + (addition). The additive identity element is labeled "0".

2. F - $\{0\}$ (the set of F with the additive identity removed) forms a commutative group under $\cdot$ (dot product). The multiplicative identity element is labeled "1".

3. The operations + and $\cdot$ distribute: $a \cdot (b + c) = (a \cdot b) + (a \cdot c)$

Galois Fields are fields that have a finite number of elements. When used in coding applications, p is usually 2 and m represents the size of the codeword. For this small example, we will use GF(2^3). The addition and multiplication tables for GF(2^3) are shown in Table 2.2 and 2.3, respectively.

Table 2.2: Addition Table for GF(2^3)

+	0	1	2	3	4	5	6	7
0	0	1	2	3	4	5	6	7
1	1	0	3	2	5	4	7	6
2	2	3	0	1	6	7	4	5
3	3	2	1	0	7	6	5	4
4	4	5	6	7	0	1	2	3
5	5	4	7	6	1	0	3	2
6	6	7	4	5	2	3	0	1
7	7	6	5	4	3	2	1	0

Table 2.3: Multiplication Table for GF(2^3)

x	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	1	2	3	4	5	6	7
2	0	2	4	6	3	1	7	5
3	0	3	6	5	7	4	1	2
4	0	4	3	7	6	2	5	1
5	0	5	1	4	2	7	3	6
6	0	6	7	1	5	3	2	4
7	0	7	5	2	1	6	4	3

For this example we will also use $n = 3$ and $k = 5$. Where n is the number of data packets and k is the number of check packets. Reed-Solomon codes must satisfy the property $n + k <= 2^m$ and therefore, only n number of packets (data or check) need to be received in order to decode the n data packets. Finally, the error correction code requires a matrix with $n + k$ rows and n columns and satisfies the following two properties

1. The first n rows form an nxn identity matrix.

2. Any n rows of the matrix are linearly independent.

Property two is especially important so that the matrix is invertible and provides the ability to decode the necessary packets.

The Vandermonde matrix can be used to derive our required matrix because it is known to follow property two from above. The general form of the Vandermonde matrix, for any given n and m is shown below

$$\begin{bmatrix} 0^0 & 0^1 & 0^2 & ... & 0^{(n-1)} \\ 1^0 & 1^1 & 1^2 & ... & 1^{(n-1)} \\ 2^0 & 2^1 & 2^2 & ... & 2^{(n-1)} \\ ... & ... & ... & ... & 0^{(n-1)} \\ (2m-1)^0 & (2m-1)^1 & (2m-1)^2 & ... & (2m-1)^{(n-1)} \end{bmatrix}$$

For our given choice of parameters, m = 3, n = 3, and k = 5, the Vandermonde matrix is as follows

$$V = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 1 & 1 \\ 1 & 2 & 4 \\ 1 & 3 & 5 \\ 1 & 4 & 6 \\ 1 & 5 & 7 \\ 1 & 6 & 2 \\ 1 & 7 & 3 \end{bmatrix}$$

Since matrix V is derived from the Vandermonde matrix, it satisfies property 2. However, we need to perform a series of linear transformations in order for matrix V to satisfy property 1.

$$D = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 1 & 6 \\ 4 & 3 & 2 \\ 5 & 2 & 2 \\ 5 & 3 & 4 \\ 4 & 2 & 4 \end{bmatrix}$$

From the transformed Vandermonde matrix, D, the bottom k rows are used to generate the k check packets. This is done by performing matrix multiplication on the bottom k rows of D and a vector of the three data packets that will be sent. For this example, let us choose our data packets to have the values $\{6, 1, 4\}$.

$$\begin{bmatrix} 1 & 1 & 6 \\ 4 & 3 & 2 \\ 5 & 2 & 2 \\ 5 & 3 & 4 \\ 4 & 2 & 4 \end{bmatrix} \times \begin{bmatrix} 6 \\ 1 \\ 4 \end{bmatrix} = \begin{bmatrix} 2 \\ 5 \\ 2 \\ 6 \\ 1 \end{bmatrix}$$

The resulting k check packets are $\{2, 5, 2, 6, 1\}$. When the information is sent across a coding channel, each packet has an associated index value with it. The following information is transmitted from the sender to the receiver.

$$\{(0, 6), (1, 1), (2, 4), (3, 2), (4, 5), (5, 2), (6, 6), (7, 1)\}$$

After the transmitted information gets sent over a coding channel, some packets may be mutated or lost altogether. The receiver only need know the values m, n and k and how to derive the Vandermonde matrix to correctly decode the sent data packets. Continuing with the given example, suppose only the check packets at index values 4, 6 and 7 are received. Therefore, we need the corresponding rows of the derived Vandermonde matrix, shown below as D'.

$$D' = \begin{bmatrix} 4 & 3 & 2 \\ 5 & 3 & 4 \\ 4 & 2 & 4 \end{bmatrix}$$

The lost data packets are denoted d_0, d_1, and d_2. As we previously showed, the check packets were produced by performing matrix multiplication on the data packets and the corresponding rows of the Vandermonde matrix.

$$\begin{bmatrix} 4 & 3 & 2 \\ 5 & 3 & 4 \\ 4 & 2 & 4 \end{bmatrix} \times \begin{bmatrix} d_0 \\ d_1 \\ d_2 \end{bmatrix} = \begin{bmatrix} 5 \\ 6 \\ 1 \end{bmatrix}$$

Solving the equation above, will uncover the correct values for the data packets. The equation can be solved by inverting D' and performing matrix multiplication on the values of the check packets. After inverting the matrix and performing the calculations, we can confirm the values of the data packets as $\{6, 1, 4\}$.

$$\begin{bmatrix} 7 & 5 & 3 \\ 7 & 4 & 2 \\ 1 & 7 & 5 \end{bmatrix} \times \begin{bmatrix} 5 \\ 6 \\ 1 \end{bmatrix} = \begin{bmatrix} 6 \\ 1 \\ 4 \end{bmatrix}$$

CHAPTER 3

RELATED WORK

Even before the need for better data storage solutions, the idea of using DNA to store information was first introduced in 1999 by Clelland, Risca, and Bancroft [14]. In the next decade, more researchers began studying the idea, and advances in DNA sequencing technologies brought on by the Human Genome Project, along with the growing need for better long-term data storage solutions, drew more researchers to the field. Between 2010 to 2018, more research was published on the topic of using DNA for data storage and the corresponding encoding and decoding algorithms. This section describes seven of the more prominent approaches in detail and includes an analysis of each approach.

The following section is organized with the intent to show advancements made in the field over time. Each approach shows some improvements and new ideas presented over previous works. These approaches are presented in the order that they were published with the exception of the technique from Organick et. al., whose paper had not been peer reviewed at the time that this research was completed. Therefore, their algorithm is included after Bornholt, et. al. because both approaches come from the same group of researchers collaborating at Microsoft Research and the University of Washington. While there are many more algorithms published on this topic, we have chosen to only include approaches that employ next-generation sequencing techniques

and have included the results of laboratory testing. The algorithms presented also address the same research questions that are discussed in this thesis. The work of this thesis presents a next step in the progression of these ideas over time by building upon the research of the most recent approach by Erlich and Zielinski [17].

3.1 Church [13]

"Next-Generation Digital Information Storage in DNA," by Church, Gao, and Kosuri is one of the first techniques to demonstrate the use of next-generation synthesis and sequencing technologies for storing large amounts of information. Published in September of 2012, their approach includes an experiment that successfully stored 5.27 megabits of data, compared to the previous maximum of 7920 bits. This ability to store more information was aided by next-generation technologies which demonstrated about 100,000-fold less cost than first-generation techniques. These new technologies were a key factor in the growth of research in algorithm development for DNA data storage.

The data that was stored, including text, images, and a JavaScript program, was converted into a bit stream and these bits then became 54,898 oligonucleotides (oligos) that were each 159 nucleotides (nt) long. Each oligo included a 96-nt data block, a 19-nt address and flanking 22-nt common sequences that are used for amplification and sequencing. More specifically, the address section determines the location of the data block in the bit-stream.

The encoding algorithm that was used is a basic mapping of bits for nucleotides. A zero is replaced by an A or C and a one is replaced by a G or T. The decision in choosing A versus C or G versus T is made by the following constraints; resulting

sequences should not contain extreme GC content, repeats, or secondary structures. The decoding stage then swaps an A or C for zero and a G or T for a one. When recovering the data, in the experiment, all blocks were recovered with a total of 10 bit errors out of 5.27 million. These errors generally occurred within homopolymer runs at the end of the oligos where there was only single sequence coverage. Insertion and deletion errors were not recorded, although, Table 3.1 shows the amount of each type of mutation. Because the encoding algorithm maps two nucleotides to one bit, only 10 of the discrepancies resulted in an actual bit error in the digital data.

Table 3.1: Church: Analysis of Mutations Found

Original	A	C	T	A	T	G	C
Mutation	G	G	G	C	C	C	A
# of times	3	2	1	8	3	2	3

Church's approach represents a promising first step in developing an encoding and decoding algorithm and a significant improvement in the amount of data that can be stored in DNA. Some amount of implicit error correction also occurs without the need for error detection. However, a main goal of data storage is to be able to recover all information without any error. A natural next step would be to include more robustness to errors that may occur during the synthesis, storage or sequencing processes.

3.2 Goldman [19]

In the development of their algorithm, Goldman and his team were looking to achieve an improvement upon the ideas presented by Grass, et. al.. A more scalable approach to DNA data storage encoding and decoding that is also more robust to errors, is detailed in their research, "Towards practical, high-capacity, low-maintenance infor-

mation storage in synthesized DNA." In order to demonstrate that their approach can work for arbitrary digital data, their experiment stored five different types of files, including text, PDF, JPEG, MP3 and ASCII. This dataset totaled 0.75 megabytes.

In pursuance of a more robust technique, Goldman and his colleagues designed an encoding algorithm that is relatively more complicated than previous designs. The algorithm not only converts digital data to nucleotides, it also pads sequences to be of a certain length, includes addressing and parity information, and takes the reverse complement of every other strand.

The first step is to convert digital data bytes to base-3 using a Huffman code. Table 3.2 shows how to convert the base-3 values to nucleotides based on the previous nucleotide, and initializing with a previous value of "A". This table ensures that there will not be repeating nucleotides in a row. Next, the nucleotide encoded strings are split into overlapping segments that are 100 nucleotides long (denoted F_i). If i is odd, the reverse complement of the strand is selected. A parity bit is also computed and appended to each strand. Each strand will include 117 nucleotides, that are broken down as follows: 1 which represents forward or reverse complement strands, 100 of data information, 2 for the file location, 12 for the address within the file, 1 for parity and 1 more to label forward or reverse complement strands. An overview of this process is shown in Figure 3.1.

Table 3.2: Goldman Mapping Scheme

previous	next trit to encode		
nt written	0	1	2
A	C	G	T
C	G	T	A
G	T	A	C
T	A	C	G

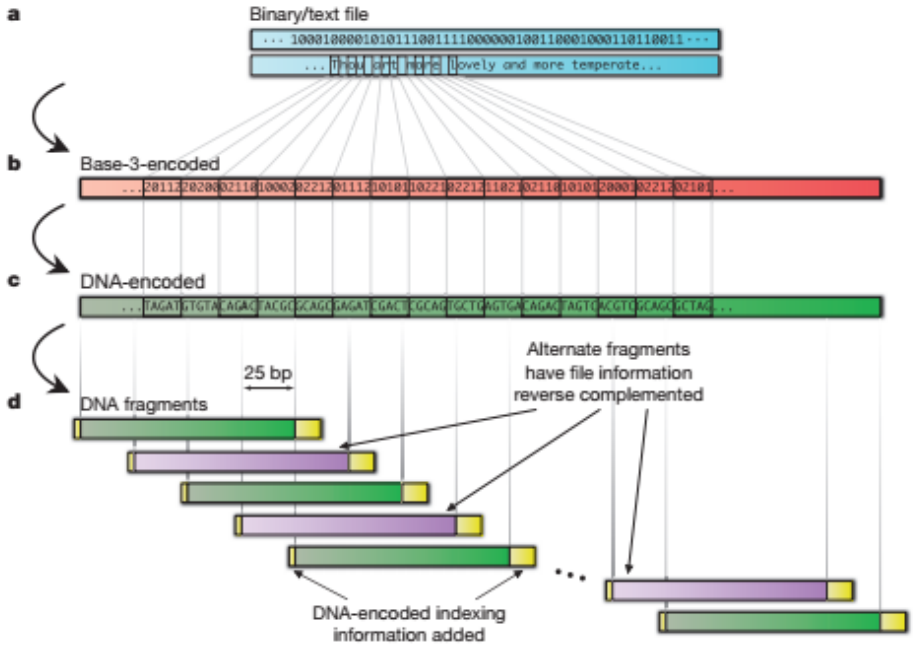


Figure 3.1: Visualization of Goldman [19] Encoding Algorithm

Even though the researchers were able to test this algorithm and recover all five digital data files with 100% accuracy, there were some issues that were encountered during the experiments. Inconsistencies that were introduced during synthesis or sequencing were discarded. More sequences that were found to have errors during the decoding process were also abandoned. These errors were likely detected by examining parity bits and the length of segments. Luckily, because of the amount of redundancy and the use of majority voting among redundant sections, there were enough sequences to recover the original information.

Another issue was that two sections, each 25 bases in length, were not able to be recovered during the experiment. These gaps, totaling 50 bases, had to be manually inserted in order to restore the original information. Goldman and his team were able to hypothesize how to fill in these segments by inspecting neighboring

regions and manually inserting bases so that the sequences could be decoded. Other inconsistencies were caused by errors within the address sections of the sequences. These types of errors misplaced strands and misconstrued the majority voting.

In a more detailed analysis of where errors were occurring, the researchers noted that the unrecoverable sequences were very likely caused by long repeating segments of 20 bases within the strands. These segments happen to be self-complementary and can form bonds causing strands to fold over on themselves. This makes it extremely difficult for the sequencing process to read the correct sequences. Typically, the severity of the issue varies depending on the sequencing technology used. In the case of this experiment, those segments were not able to be sequenced at all. In any case, having an algorithm that avoids creating longer repeating or self-complementary sequences is a good idea and was admitted in hindsight by the researchers.

Attention should be paid to these types of errors, and the data that might lead to bad sequences which cause them, especially in the addressing section. This is because an algorithm, like this one, indexes strands numerically and then pads the length to be constant. Having a wide range of index values that starts with small numbers could generate sequences that are padded with many leading zeros. Based on the mapping scheme, these repeating zeros could potentially generate long repeating sequences of nucleotides that might then cause issues for the synthesis or sequencing technology. This issue may become more relevant as these algorithms and techniques become more scalable and adapt to store larger amounts of data, as is demanded by industry.

This algorithm has many improvements over previously presented designs. It is scalable and the use of a parity bit, as well as four-fold redundancy, makes it more robust to errors. It is difficult to overlook the fact that 50 bases had to be manually reconstructed to recover one of the files, but it brings attention to the issue of avoiding

long repeating and self-complementary sequences. Succeeding algorithms attempt to address this point, as well as including more vigorous error-correction techniques.

3.3 Grass [20]

In building upon previous ideas, Grass and his colleagues wanted to incorporate a specific technique for error correction in an encoding and decoding algorithm. Their design is presented "Robust Chemical Preservation of Digital Information on DNA in Silica with Error-Correcting Codes." One of the main areas of focus for this work was to implement a strong error correction model in place of none, or simple redundancy.

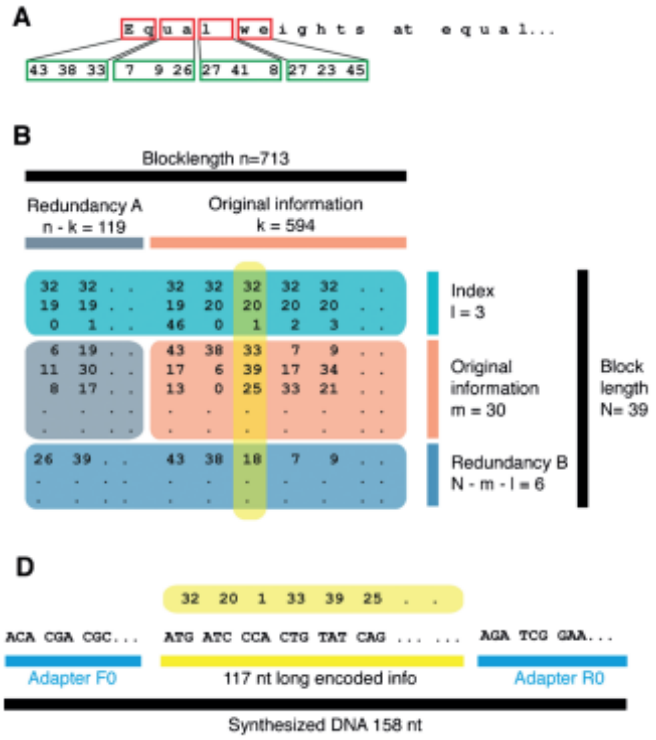


Figure 3.2: Visualization of Grass [20] Encoding Algorithm

For the error correction model, the researchers incorporated two levels of Reed Solomon error correction codes into their encoding and decoding algorithm. The digital data is first converted into a number within the Galois Field of size 47 (GF(47)). These numbers are then put into a block of 594 x 30 values. This can be seen in the orange block in section B of Figure 3.2. The first Reed Solomon parity information is added on each row, in the form of 119 values from the GF(47). This section is referred to as the outer block, or Redundancy A, and is shown in the gray section of the diagram. Next the teal section is added, which represents 3 values of indexing information. Also appended to each column is the second level of Reed Solomon parity. This section is represented by the blue section of the diagram and is referred to as the inner block, or Redundancy B. The amount added to each column for Redundancy B is 6 values long. Each column now consists of 39 values (one is highlighted in yellow in the diagram) and is converted to nucleotides using a special mapping scheme.

The wheel shown in Figure 3.3 represents the mapping scheme and converts each value of the GF(47) to a sequence of three nucleotides and is designed in a way that will not produce runs of more than three of the same nucleotides in a row. The end result is that each strand consists of 117 nucleotides and is flanked by constant adapters for a total strand length of 158 nucleotides.

The decoding of the algorithm works by first converting the nucleotides of each strand back to the values of the GF(47) using the same wheel diagram. Strands are then put in order based on the indexing information. At this point, any errors that may have occurred, can be corrected by the inner Reed Solomon code. It is also possible that a certain number of strands will be missing altogether. If this is the case, the outer Reed Solomon code should be able to correct the remaining errors.

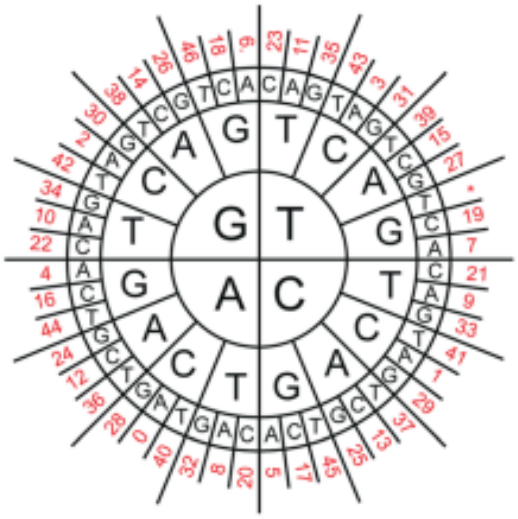


Figure 3.3: Grass [20] Mapping Scheme

Once this step is complete, the data block of the GF(47) values can be converted back to the original digital data.

Grass and his colleagues tested this algorithm by storing two ancient documents in DNA. They put the Swiss Federal Charter from 1291 and the English translation of the Method of Archimedes through their algorithm in order to determine the 4,991 sequences that would be used to construct DNA strands. Upon sequencing and decoding the resulting sequences, they found that the Reed Solomon codes were successful in detecting and correcting errors that occurred during the synthesis and sequencing processes. They determined that the inner code had to correct an average of 0.7 nt errors per sequence. The outer code had to account for a loss of 0.3% of total sequences and correct about 0.4% of the sequences.

In a more detailed analysis of the error correction capabilities of this technique, the researchers calculated that the inner code can handle at least three individual base errors per sequence. In general, the estimated error rate per nucleotide that can

occur during sequencing is 0.1%. Therefore, correcting three base errors in a strand of length 117 nts should be sufficient to recover from the mutation errors that may occur. Insertion and deletion errors were disregarded from the analysis. If one of these types of errors did happen, the system could detect that the segment length was incorrect and drop those segments before moving on to the next level of Reed Solomon error correction. The researchers figure that the outer code can handle the loss of about 17% of all complete segments, or correct about 8.5% of all complete sequences if they contain errors after the inner error correction code was applied. This indicates that the algorithm would indirectly be able to recover from a certain amount of insertion or deletion errors, in addition to correcting mutations. More research would need to be done to directly detect and recover from multiple insertion and deletion errors. This leads to a final consideration of this design.

The data in this system is highly coupled with the error correction codes. This implies that all of the DNA strands must be decoded together in order to retrieve any one section of information. At this point in the field, as algorithms have become more scalable and better at handling errors, researchers must consider how to access different sections of the data. Random access of, and varying techniques for indexing information are the next set of obstacles to overcome that the algorithm presented in this research had not yet considered.

3.4 Bornholt [10]

New ideas for indexing and storage techniques were brought to light by Bornholt and his colleagues in their research from 2016, "A DNA-based archival storage system." The two main contributions include a new architecture for DNA data storage that

allows for random access of data, as well as a new encoding scheme that offers controllable redundancy. These ideas are supported by experiments and simulations on a data set of four different images, totaling 0.15 megabytes.

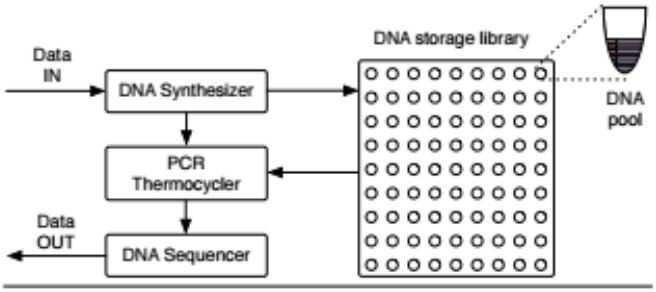
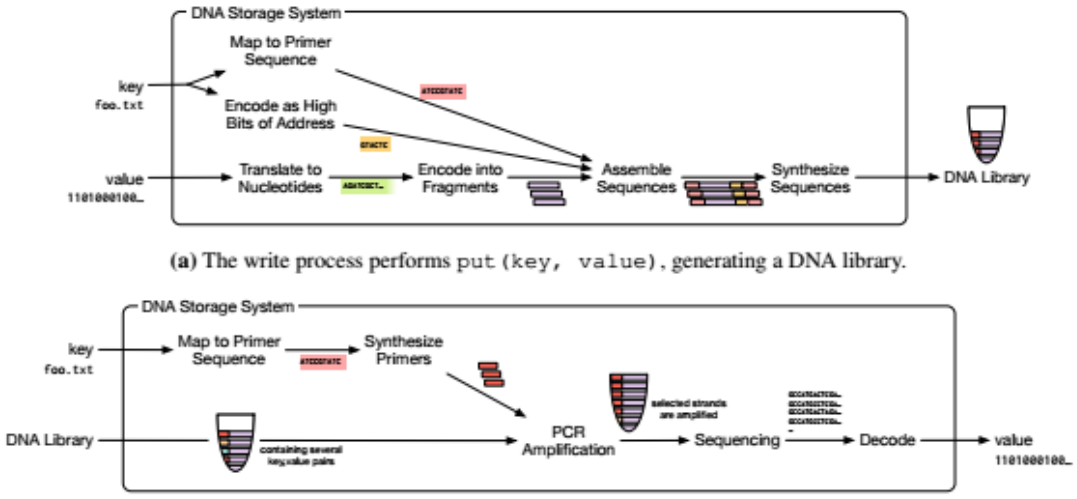


Figure 3. Overview of a DNA storage system.

Figure 3.4: Bornholt [10] Pooled Architecture

The first contribution presented in this research is the new architecture. The authors of this algorithm demonstrate an idea for storing DNA in separate pools. The algorithm then incorporates a (key, value) system in order to store and access different sections of the data set from the corresponding pools. The key part of the system might represent the name of a particular file that a user would want to access. This key would then map to a specific PCR primer sequence, where the primer sequence would consist of two parts. The first part of the primer represents which pool the DNA might come from and the second part is the position of that data section, within that file. The value associated with the key is the DNA sequence representation of the data, along with an error correction section. In order to read the information back, the key would be mapped to the corresponding primer sequence. Then that primer sequence can be used to extract the relevant strands needed to retrieve the data.



(a) The write process performs `put(key, value)`, generating a DNA library.

(b) The read process performs `get(key)` on a DNA library, returning the value.

Figure 4. Overview of a DNA storage system operation as a key-value store.

Figure 3.5: Example of Bornholt [10] (Key, Value) Encoding

This architecture represents a trade off between reliability, performance, and storage density. In a comparison between this design and one that stores all strands in one pool, this architecture is much less dense in terms of the physical density needed for the system. Other issues can be minimized, for example, using multiple pools lessens the chance that different primers may react poorly with each other. Bornholt and his team also argue that having a single pool reduces the likelihood that a random sample drawn would contain all of the desired strands. The system may need to make several reads from the pool in order to retrieve all of the pertinent information. With more pools and a higher likelihood of gathering all the needed strands in one step, this would arguably save time and energy. The experiments in this research were therefore made to use "reasonably-sized" pools. This value could potentially be varied by the user to maximize performance and storage density for a

given system and in adaptation to changing trends in the industry.

One other advantage of this design is that different encoding algorithms could be used for the actual conversion or mapping from digital data to nucleotide sequences. In order to test the idea, the authors of this technique employed the Goldman algorithm for encoding and decoding their data along with the (key, value) system and a new algorithm. Because of the flexibility provided by the architecture design, Bornholt and his team were able to experiment with another algorithmic idea for mapping that provides a lower level of redundancy. While the Goldman algorithm incorporates overlapping strands with a four-fold redundancy, a user may determine that level of redundancy is not required. The researchers even suggest that different sections of data could be encoded with different levels of redundancy, depending on the importance of the data.

The newly presented mapping idea, that incorporates less redundancy, is called XOR encoding. Under this plan, two DNA strands, comprised of data, are combined with the exclusive-or function to generate a third strand representing the parity strand. A diagram of this idea is shown in Figure 3.6. Ideally, only two of the three strands would be needed to recover the third strand. This would work well for recovering entire strands if they are lost, or even fixing insertion or deletion errors. Mutation errors could be detected in the system but would be more difficult to correct. If any inconsistencies were found when computing the exclusive-or functions, there would be no way to determine which strand the mutation came from. An extra level of error detection on each individual strand would likely be needed to uncover where the mutation was located.

A few different experiments and simulations were completed to test the different algorithms. The Goldman algorithm, as well as the XOR encoding algorithm were

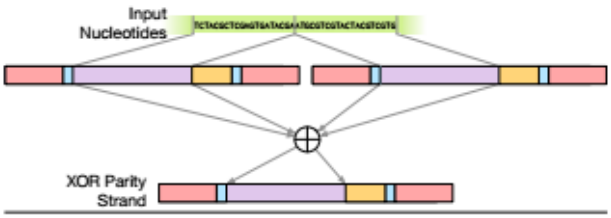


Figure 8. Our proposed encoding incorporates redundancy by taking the exclusive-or of two payloads to form a third. Recovering any two of the three strands is sufficient to recover the third.

Figure 3.6: Visualization of Bornholt [10] XOR Redundancy

used with the new architecture idea on four different image files. The files varied in size from 5kB to 84kB. Combined, these eight files produced 45,652 sequences, each 120 nucleotides long. To demonstrate the random access technique, three of the four Goldman encoded files were recovered and one of the four XOR encoded files was recovered. The experiments demonstrated a successful result in the ability to only amplify the targeted files and did not retrieve sequences from the other files. In recovering and decoding all of the files, a one-byte error was found in one file, encoded with the Goldman algorithm, that needed to be fixed by hand.

Testing the XOR encoding algorithm allowed the researchers to conduct more analysis on the DNA data storage system by taking a subset of the produced strands. Since the XOR encoding produces both parity strands and data strands, dropping the parity strands would represent a naive encoding that does not include any redundancy. In decoding and analyzing just the naive encoding set, 11 DNA strands were missing entirely. A valid JPEG file could not be recovered without the missing information. The results of this experiment verify the need for some level of redundancy or error correction to be included in the process using DNA for data storage. This holds true whether or not the system uses a single or multiple pooled architecture.

3.5 Organick [29]

This next algorithm, described in "Scaling up DNA data storage and random access retrieval," comes from the same group that worked on the previously described algorithm. In order to expand on the ideas previously presented, the authors of this technique had the goal of scaling up their experiments in terms of the amount of data successfully stored in DNA. Another improvement on these experiments was to incorporate a more robust error correction scheme, instead of varying levels of redundancy, which can be seen in the form of a Reed Solomon code. Along with these new goals, experiments included tests for random access retrieval in order to incorporate previous work.

At the time of this research, the authors claim that they had stored the most data, to date, in DNA. Their data set includes a music video from OK Go (HD video), consisting of 44.2 MB and 3.2 million sequences, a classical music collection (music), consisting of 13.9 MB, and 890 thousand sequences. The Crop Trust database of the seeds stored in the Svalbard Global Seed Vault (text), consisting of 11.1 MB, and 708 thousand sequences, and the Universal Declaration of Human Rights in over 100 languages. The total amount of data stored is 200.2 MB, and 13.4 million sequences of DNA.

Important elements of this algorithm include the use of an exclusive-or operation, a Reed Solomon parity code and a selective amplification technique to successfully implement random access of the data set. The researchers also designed and validated a library of primers in order to make the experiments possible.

A high-level overview of the encoding algorithm can be seen in section b of Figure 3.7. First, the exclusive-or operation is used to combine the input data with the

output of a seeded pseudo-random number generator. This step helps to break up multi-bit repeats that may be found in the input data. Next, the randomized data is partitioned into Reed Solomon codewords. Then, the bit sequences are converted to nucleotide sequences with a rotating code that eliminates homopolymers. While the rotating code is not explicitly described, one might assume that it would be the same or similar to what is used by the Goldman algorithm, since that algorithm was used in previous research by this group. Finally, an address that describes the sequence's location, in relation to the other sequences is added to each sequence, along with a 20-base primer sequence on each end.

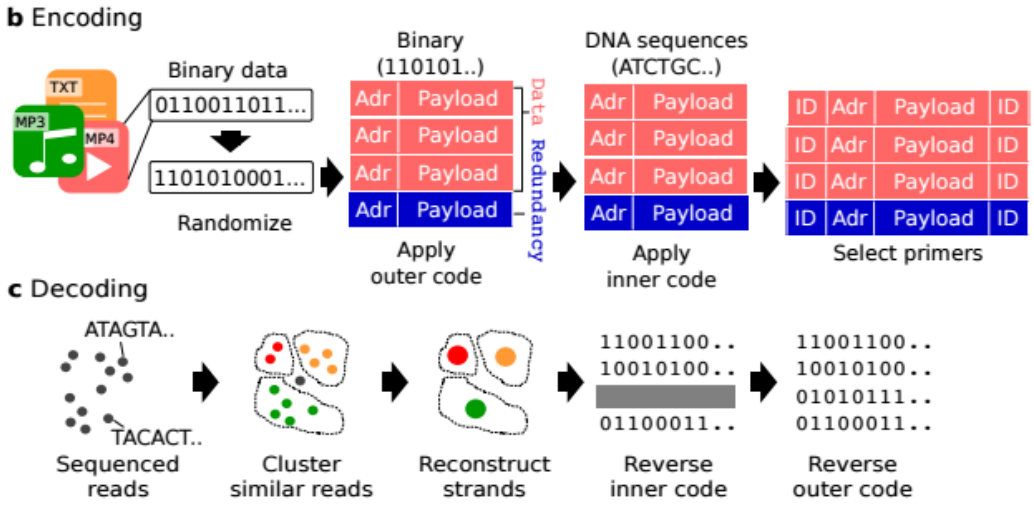


Figure 3.7: Visualization of Organick [29] Encoding and Decoding Algorithms

With the new elements of the encoding algorithm, the decoding algorithm was also updated. The decoding algorithm can be described in four stages as is shown in section c of Figure 3.7. In the first stage, strands are clustered by the similarity of their entire content, not just the address. This is done with an "algorithm based on an iterative locality sensitive hashing". This also means that strands aren't required

to be of a certain length. Other algorithms studied typically disregard strands that are of incorrect length. Attempting to correct those errors allows the algorithm to use the rest of the information in the strand so it is not lost. Part two uses a variant of the bitwise Majority Alignment algorithm which is adapted to support insertions, deletions and substitutions. This is used to process each cluster to recover the original sequence, based on a variant of a majority voting technique. In part three, strands are decoded from nucleotides to bits in order to obtain indexes and values of individual coordinates of the outer Reed Solomon code. In part four, the outer Reed Solomon code is used to correct any errors that may have occurred. Finally, the exclusive-or randomization is inverted to discover the original input data.

In expanding upon previous works, this research showed many improvements. Organick et. al. completed experiments that stored the most data in DNA than has ever before been seen. In terms of algorithm development, they have incorporated some previously seen techniques into their algorithm in new ways. Employing Reed Solomon codes for error correction has been implemented before, but not in conjunction with random access techniques. Although it was not explicitly described in their research, the Reed Solomon codes must have been broken up among different files or sections in order to decouple all of the data from the parity information. This would allow random access of specific sections of data without having to decode the entire data set. The same way of breaking up the data might also be used for the exclusive-or operation with a randomly seeded value. If one value was used for the entire data set then it would be difficult to line up a portion of that for the subsection of data that was to be accessed individually.

Whether or not one or multiple randomly seeded values are used, the seed would at least need to be stored outside of the DNA data storage in order to be used for

decoding later on. While a user may not want extra information to retain, the idea of using the exclusive-or operation to randomize the data provides a solution for the problem encountered by Goldman et. al. Their design ran into issues in dealing with long repeating sections of information and the difficulty in handling this issue could be lessened with an exclusive-or operation with a randomized value.

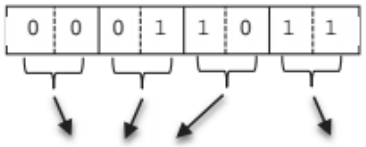
3.6 Blawat [9]

The algorithm ideas presented in "Forward error correction for DNA data storage," represent one of the most complex encoding and decoding algorithms that have been researched and tested. Not only is a new method for error correction described, different levels of error correction are also incorporated throughout the different steps of the encoding algorithm. This seems to provide a way to more accurately recover mutated or altered DNA strands that is extremely robust.

More specifically, there are three different types of error correction that are employed during different steps of the encoding algorithm. The first type of error correction comes in the form of a BCH code that is used on the address portion of a given sequence. Also appended to each strand is a 16 bit Cyclic Redundancy Check (CRC). The CRC provides error correction over the data section of the strand. The final level of error correction is a Reed Solomon code that is applied over an entire data block. These three types of error correction are incorporated while the data is in the form of bits. After the data has been broken into sections, given an address, and had error correction applied to it, the sequences of bits are then converted to nucleotides.

The part of the encoding algorithm that converts bits to nucleotides employs two

different mappings. Table 1 of Figure 3.8 displays the first mapping which converts two data bits to one nucleotide symbol. This mapping is a direct mapping from one two-bit set to one nucleotide. The second mapping, represented by Table 2 of Figure 3.8, maps two data bits to a two-nucleotide set. Each two-bit set can be converted to one of four different options of nucleotides sets.



Value	Nucleotide
00	A
01	C
10	G
11	T

Table 1: Mapping of first 3 bit-tupel to 3 nucleotides

Value	Option 1	Option 2	Option 3	Option 4
00	AA	CC	GG	TT
01	AC	CG	GT	TA
10	AG	CT	GA	TC
11	AT	CA	GC	TG

Table 2: Mapping of last 2 bits to a pair of nucleotides

Figure 3.8: Blawat [9] Mapping Tables

These two mappings are used to convert 8 bits to 5 nucleotides at a time. The first and second bits, map to the first nucleotide of the converted sequence, using the first table. The third and fourth bits map to the second nucleotide and the fifth and sixth bits map to the fourth nucleotide, also using the first table. Then the last two bits (seventh and eight), are converted to the third and fifth nucleotides by using the second table. Since the second table maps to two nucleotides, the two are split up and the first of the pair goes into the third position of the final sequence and the second of the pair becomes the last nucleotide (fifth) of the final converted sequence. Which nucleotide pair that is chosen out of the four options from table two is chosen in order to avoiding any repeating nucleotides in a row.

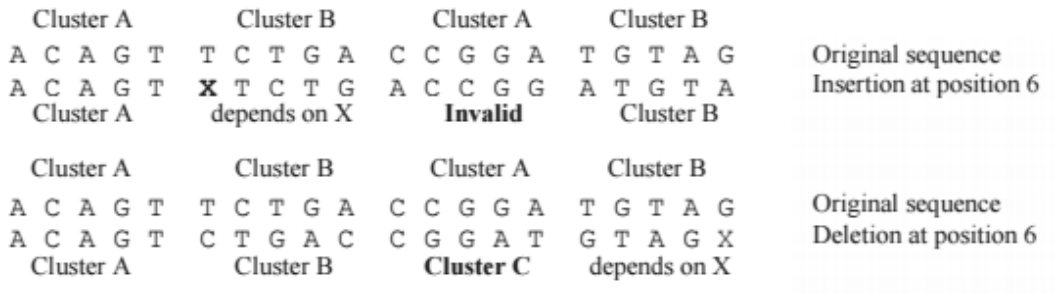


Figure 3.9: Blawat [9] Error Detection Technique

The decoding portion of the algorithm can be described in four main stages. Each stage pulls from the pool of recovered sequences and leaves the rest to be considered in later stages. In the first stage, the only strands that are considered are the ones that have the correct length, pass the CRC, have a correct BCH code on the address and can be converted back to bits without error. Stage two examines strands that are the correct length but have an erroneous CRC or are greater than or less than the correct length by one nucleotide. For strands with an incorrect CRC, those with the same address are grouped together and majority voting is used to determine the sequence to be kept. For the strands that are off by one, in terms of length, a single insertion or deletion error likely occurred. The algorithm that converts the nucleotides back to bits attempts to find these errors and correct them by deleting or inserting single nucleotides until a valid codeword is generated. An example of this technique is shown in Figure 3.9. Next, strands that are significantly too short are analyzed in stage three. For these, majority voting is used at each nucleotide position to piece together sequences that can complete the entire strand. Finally, in stage four, the Reed Solomon error correction code is applied over the sequences found in stages one through three. This step can recover entire strands that were not successfully recovered, as well as correct any sequences that were decoded incorrectly.

In an experiment that stored 22 Megabytes of data in DNA, the authors of this research explained that they were able to use their encoding and decoding algorithm to recover the data without errors. The experiment included four different libraries, each containing 225,000 strands of DNA. Library one had the most errors and after the decoding stages one through three, the Reed Solomon code had to recover 583 strands, or 0.26% of the overall data. In an analysis of the data, the researchers observed a correlation between the data stored in a strand and the probability that no correct copy of that strand was found. They detected that sections of data with long sequences of identical data bytes were encoded into nucleotide sequences with repetitions of ten nucleotides. These repetitions then propagated into strands that were lost or unable to be sequenced fully.

Blawat and his team introduced some novel ideas in terms of incorporating multiple error correction methods into an encoding and decoding algorithm. They also presented a new way to convert data bits into nucleotides that avoids repeats and has built in methods to discover insertion and deletion errors. Yet, one disadvantage of the mapping scheme is the types of nucleotide sequences that can be produced. Sequences of three repeating nucleotides in a row, as well as longer sections of repeated nucleotide sequences can still occur and cause problems in fully sequencing and recovering the data. Perhaps the researchers were following an approach that focused more on correcting errors than minimizing the possibility of them arising in the first place.

This is a valid strategy but brings up other constraints that must be considered in analyzing the appropriateness of a DNA data storage system. One of these considerations is computational running time. One technique this decoding algorithm employed was to replace each nucleotide one by one, to find a valid sequence, when attempting to correct insertion or deletion errors. Depending on the size of

the sequence and the number of errors, this could be a slow process in terms of computing time. Another constraint to consider is the information density of the DNA. Creating more strands that include parity information or error correction codes lowers the information density over the total amount of DNA produced and increases the amount of DNA needed for a given amount of data. Because of the currently high cost of production, making more DNA strands is not trivial. Establishing a necessary amount of error correction that balances information density and accuracy in recovering from imprecisions is a significant challenge that requires more research and experimentation.

3.7 Erlich [17]

The ability to approach the information capacity per nucleotide while providing robustness towards correcting errors is the goal of the algorithm described in "DNA Fountain enables a robust and efficient storage architecture." By employing a Fountain Code, this allows the algorithm to approach the Shannon Information capacity. Experiments described in this research show that the authors were able to store 2.15 MB of varying digital data, including an operating system and a short movie, and recover the information completely with no errors.

The encoding part of the algorithm comprises of three main stages, including preprocessing, the Luby Transform, and screening. A diagram outlining the process is shown in Figure 3.10. The first step of the preprocessing stage includes compressing the data files. This step helps to minimize the amount of data encoded as well as removing repeating bits for more randomized data. Next, the digital data is split into non-overlapping segments of length L. This length, L, can be adjusted by the user,

and determines how long the final nucleotide strands will be. The researchers decided upon segments of length 32 bits, which eventually produced strands with a length of 200 nucleotides.

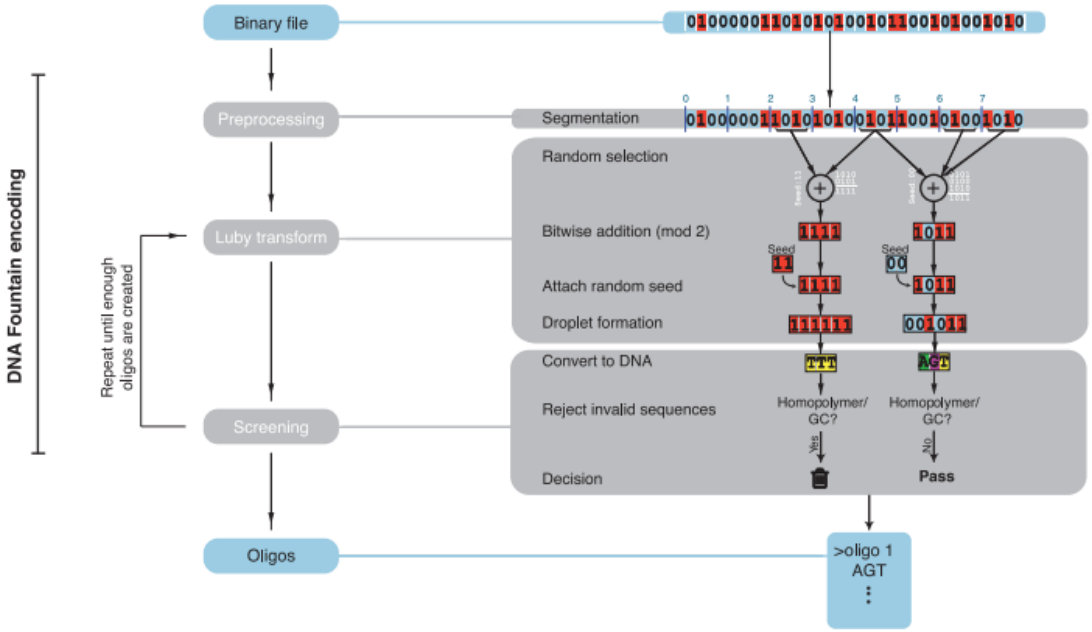


Figure 3.10: Visualization of Erlich [17] Encoding Algorithm

After preprocessing, the encoding algorithm moves on to the Luby Transform stage. This step includes creating sequences of bits called droplets. Each droplet includes a seed value and has the option to include an error correction code. In their experiments, Erlich and Zielinski used a Reed Solomon code attached to each droplet. The seed values, which can also be considered an index value, are used to initialize pseudo random number generators (PRNG) and are generated in a specialized way. A standard seed generation for a Fountain Code would start at zero and iterate by one to the maximum value produced by the bit size. However, for this algorithm, the typical seed generation would produce bursts of homopolymer runs which would either need

to be excluded, or would cause problems for the synthesis and sequencing processes. Instead, the algorithm employs a Galois linear-feedback shift register (LFSR). This allows the algorithm to iterate over each number in the interval [1,...,2^{32}-1] in a pseudo-random order and without repetition.

Once a seed is generated for a droplet, two PRNGs are initialized with that seed. The first PRNG is created over a Robust Soliton probability distribution, shown in Figure 3.11, and is used to choose *d*. This value, *d*, represents how many data segments, of length L, will be chosen for the droplet. The second PRNG is created over a uniform distribution and is used to select those *d* different segments, without replacement, from the initial digital data. To finalize each droplet, the algorithm takes the exclusive-or operation over the *d* chosen segments and the outcome becomes the data payload for the droplet. The seed and optional error correction code are then appended to the droplet. More information about the Luby Transform stage and the Robust Soliton distribution can be found in Section 4.2.

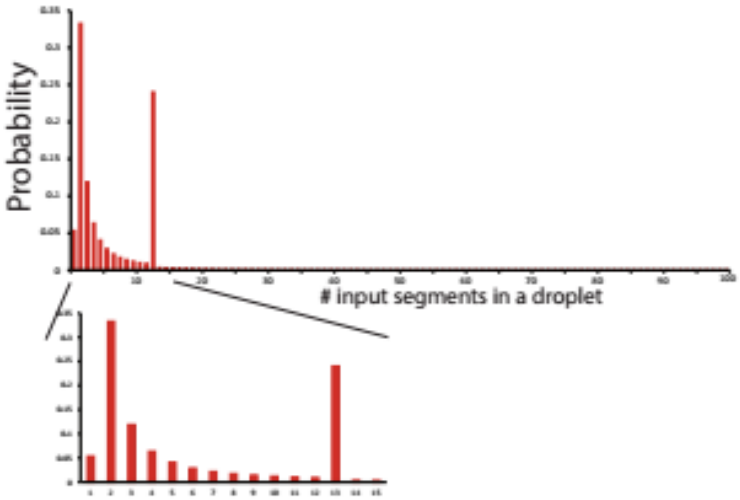


Figure 3.11: Soliton Distribution for the Luby Transform [17]

The final stage of the encoding algorithm is the screening stage. At this stage, the bits are converted to nucleotides using the following mapping: {00,01,10,11} to {A,C,G,T}. The algorithm can then analyze each droplet in the form of a sequence of nucleotides. Sequences that do not conform to certain properties are then discarded and not used to generate DNA strands. Valid sequences that are kept are ones that meet the requirements for GC content and homopolymer runs. For this experiment, a GC content level between 45 to 55% and length of homopolymer runs less than or equal to 3 nucleotides was used in order to screen the droplets. Since sequences can be dropped in this stage, the algorithm has the option to go back to the Luby Transform to continue to produce droplets until the desired number of nucleotide strands are accepted. This value, the number of oligonucleotides produced, can be determined by the user depending on the preferred level of redundancy. For this technique, Erlich and Zielinski generated 72,000 strands, which yielded a redundancy of 7%.

The decoding part of the algorithm also consists of three major stages which are preprocessing, droplet recovery and segment inference. In the preprocessing stage, nucleotide strands are collected and ordered. Only strands of the correct length are kept. Strands of incorrect length would indicate insertion or deletion errors and are discarded. Then, the remaining strands are grouped and ordered by the number of occurrences within each group. This allows the algorithm to first process strands with multiple occurrences that are less likely to have errors. Singletons, which have a higher probability of including errors can be left to be processed last, if at all.

The second stage of the decoding algorithm is the droplet recovery stage. At this point, nucleotides are converted back to binary with the same mapping scheme used to convert bits to nucleotides. Once in the binary form, the sequence can be separated into the seed, data payload, and error correction parts. The error correction

can be applied to the data and if any inconsistencies are detected, then the segment is discarded.

For the third and final stage of the decoding algorithm, termed segment inference, the recovered droplets are used to uncover the original digital data. As is done in the Luby Transform stage of the encoding algorithm, two PRNGs are produced with the recovered seed from each droplet. The first determines how many segments are included in the droplet and the second produces the input segment indexes. The algorithm now knows which segments are included in each droplet and must apply the exclusive-or operation to recover each segment. When one segment is left in a given droplet, that bitstring is assigned to the corresponding segment index (in the final recovered data) and the value is propagated throughout the rest of the droplets that also contain that segment. This process is repeated, analyzing more and more sequences, until all of the original data can be recovered.

Upon first glance, the presented algorithm seems complicated. While there are many components to understand, a survey of the details described in each step brings together an overall picture of how this algorithm works. The error correction capabilities appear to be robust enough to recover from insertion, deletion, or mutation inaccuracies without considerable amounts of redundancy. In a chart that compares their algorithm against previously published ideas, Erlich and Zielinski's design produces the highest net information density of 1.57 bits/nt. The second highest amount is only 1.14 bits/nt produced by the Grass et. al. design.

Another advantage of this algorithm is the flexibility offered to the user. Depending on the needs of the system, multiple parameters can be optimized to best meet certain needs. For example, the length and amount of the DNA strands can be altered, certain constraints can be followed or ignored when screening for valid sequences, and

the type of error correction code used on each strand could be switched, augmented, or even left out entirely. The one thing that is missing from this algorithm is the ability to access any one portion of the dataset at a time. Since information is spread across multiple droplets, all strands would need to be decoded in order to retrieve any section of the encoded data and random access is likely not possible with this system design. Even so, the flexibility of this algorithm idea seems promising.

The research presented in this thesis builds upon the ideas of the previously described algorithms, specifically the last one. Overall, some of the previous designs suffered from repetitions in the data that propagated to repetitions of nucleotides in sequences that were difficult for the synthesis and sequencing processes. This, in turn, lead to unrecoverable sections within the datasets. Some algorithms also suffered from a lack of error correction or redundancy that could successfully recover from insertion, deletion, or mutation errors. The lessons learned from these researchers influenced the design of a new algorithm, detailed in the next chapter.

CHAPTER 4

METHODOLOGY

In this chapter, we present REDNAM (Robust Encoding and Decoding of Nucleic Acid Memory), our work on creating encoding and decoding algorithms for use in DNA data storage. The encoding algorithm can be split into two complementary sections. One section defines the way that binary information is mapped to the quaternary molecules of DNA (A, T, C, G). This stage is defined as the mapping and translation stage. The other section details the rest of the operations needed to encode information, besides mapping. This stage can include the steps necessary to include an addressing scheme as well as redundancy or parity information such as an error correction code.

The first contribution of REDNAM includes a novel mapping scheme and translation process. This stage accounts for certain biological constraints that can cause errors during the synthesis and sequencing processes. This mapping and translation scheme is then integrated into a general encoding algorithm known as Fountain Codes.

4.1 Data Translation with Hex-to-Codon Mapping

The work on this project begins with the development of the mapping scheme. On the digital side, we chose hexadecimal characters which consists of 4 bits in base 16. On the other side, each hexadecimal value is converted to a codon, or a sequence of three

nucleotides. This mapping scheme draws inspiration from nature where ribonucleic acid (RNA) strands are decoded by ribosomes three nucleotides at a time, and each set of three nucleotides, or each codon, represents a different amino acid needed to construct certain proteins. By using codons, as opposed to one or two nucleotides, there are more sequences available for the mapping scheme (a total of 64 codons). The use of hexadecimal characters allows the algorithm to encode any form of data, by looking at its binary representation, as opposed to early algorithms which were restricted to textual data [8, 14, 22, 30].

When building a map that converts hexadecimal data to codons, certain types of constraints can be built into the mapping. For example, it is possible to construct a map that automatically excludes the possibility of building sequences that have more than 4 repeating nucleotides in a row by simply excluding the codons with three repeating nucleotides (AAA, TTT, CCC, and GGG) from the mapping.

When building this map that converts hexadecimal data to codons, we found that we could integrate some of the biological constraints into the structure of the map itself. For example, the encoded sequences should not have multiple repeating nucleotides in a row. Therefore, the map should not include the codons with three repeating nucleotides (AAA, TTT, CCC, and GGG). This automatically removes the chance of having sequences with more than 4 repeating nucleotides in a row.

Along with those 4 codons, 12 other codons were discarded from the map for biological purposes. These 12 codons are known as start codons and are listed in Table 4.1. During the translation process, in which ribosomes create proteins, specific start codons signal the ribosomes to start processing. When generating artificial DNA, our algorithm excludes these start codons from the sequences. This is useful in the case that artificial data DNA comes into contact with any biologically active translation

system. The results of this type of interaction are unknown.

Table 4.1: List of Start Codons Excluded from Map

AAT
ATA
ATT
ATG
CAC
CAT
CAG
CTT
CTG
TAT
TTG
GTG

This leaves 48 remaining codons to choose from for the mapping scheme. However, we do not want any two codons selected for a sequence to contain any of the 16 removed sequences in the overlap of their concatenation. In order to disallow this situation, we carefully constructed the mapping by looking at all possible codon options for a given hexadecimal character that can be followed by all 16 hexadecimal characters. That way, any two consecutive hexadecimal characters are able to be mapped to two codons in which the encoded sequence of those two codons does not contain a bad codon sequence. For example, for any given hexadecimal character, it may be mapped to a certain codon. Then it should be possible for another codon to follow the first one for each hexadecimal character. This means that there must be a valid codon option to follow the first one that would not contain a sequence that matches any of the 16 omitted codons.

In following that rule, we discovered that certain sequences of characters could not be avoided by including some codons. For example CAT, CAC, and CAG are all codons in the list of start codons to avoid. So any codon that ended with CA could

only be followed by a codon that started with an A, but not AA. After excluding codons that started with AA and any other codons in the list of excluded codon sequences, there were only 8 possible codons that could follow any particular codon that ended with CA. These 8 possible codons are not enough to give an option for each of the 16 hexadecimal characters in the mapping scheme so the four codons, starting with CA, had to be excluded from the final mapping scheme.

Table 4.2: List of Other Codons Excluded from Map

ACA	TCA	CCA	GCA	TGA	TGT	TGC	TGG	GAT

A total of 9 codons were discarded from use in the map for similar reasons and are listed in Table 4.2. While these codons could not be used in the mapping, they are allowed in the final encoded sequences. Unlike the first 16 codons which were removed, the set of 9 do not represent sequences that would contradict the biological constraints. However, including them in the map is not feasible. Therefore, the final mapping scheme includes 39 codons. This means that each hexadecimal character has either two or three options of codons to choose from during the data encoding process. Table 4.3 shows the final hexadecimal-to-codon mapping scheme used in our algorithm.

The ability for each hexadecimal character to convert to one out of a set of codon options is a specific technique for our encoding algorithm. This key feature allows the mapping scheme to account for the biological constraints detailed in Section 2.2. The ability to consider all constraints is not integrated directly into the mapping scheme. However, using this map allows for the consideration of more constraints during the translation phase of the encoding scheme. Since there are two or three options of codons to choose from during translation, the system can strive to construct optimal

Table 4.3: Hexadecimal to Codons Mapping Scheme

Hexadecimal	Codons
0	AAC, GAC
1	AAG, GAG
2	AGG, GTC, TCT
3	TCG, CGA
4	ACT, GCT, TCC
5	ACC, GCC, CGT
6	ACG, GCG
7	AGA, GGA
8	AGT, GGT
9	AGC, GGC, CCG
a	GAA, CGG
b	TAA, CAA
c	TAC, CCT, ATC
d	TAG, CGC
e	TTA, CTA, GTT
f	TTC, CTC, GTA

sequences.

During the translation phase, the algorithm works to convert each hexadecimal character into codons. At the start, a random codon is chosen from the set given in the mapping scheme. Then, for the next hexadecimal character, the algorithm attempts to choose a codon from the corresponding set. The algorithm must choose a codon that, when concatenated on the end of the sequence, conforms to each of the biological constraints. If the algorithm does not find a valid codon to continue the sequence, it must backtrack and pick a new codon for the previous hexadecimal character.

Backtracking may be needed in accounting for two other biological constraints. The first constraint is excluding longer repeating sequences from a strand. Since the map uses codons, three codons, or nine nucleotides, is the smallest length of a

sequence that we can limit to be seen one time in the entire strand. If any sequence of three codons is seen a second time in the translation stage, the algorithm will backtrack and choose a new codon. This stage also accounts for the ratio of GC codons to AT codons. As was discussed in Section 2.2, a given sequence should have close to 50% GC content. If a strand goes too far outside of the range of acceptable GC nucleotides, the algorithm will again backtrack and choose new codons.

Although this part of the algorithm allows for backtracking, the total number of times that the system can backtrack for any given sequence should be limited. After a few runtime tests, it was decided that an optimal number of backtracks per sequence is 500. This is typically enough time for the algorithm to find a valid sequence. Allowing the number of backtracks to increase would allow the algorithm to spend a long time in converting a difficult sequence. In the end a valid sequence may not even be possible and the time was wasted. As is shown in Section 5.2 on most test files, our algorithm can translate 89% of the sequences. By doubling the number of allowed backtracks, to 1000, the percentage of translated sequences only increased to slightly above 90%. Therefore, an increase in the number of backtracks allowed during the translation process would barely increases the percentage of translated sequences while largely increasing the running time.

Two constraints are satisfied with backtracking and another two are satisfied by the map itself. These next two constraints are the exclusion of repeating nucleotides and start codons. In our code implementation, when the map is used to convert a hexadecimal character, it also knows what the previously chosen codon was. Therefore, the mapping chooses the codon for the current hexadecimal value based on the previous codon. Since the map was developed to always have a codon option, a valid codon can be chosen the first time. This valid codon will not cause a sequence of three

or more repeating nucleotides and it will not form any of the start codons. In using this mapping scheme during the translation phase of the encoding, the algorithm can account for a total of four different biological constraints.

The python code below shows the main while loop that translates hexadecimal values to codons.

```
while i < hex_index:
  if (num_backtrack > tot_allowed_back or i==0):
  # taking too long for backtracking
  return -1


  char = hex_sequence[i]
  options = map_obj.get_codon(prev_codon, char)


  if prev_choices[i] == len(options):
    # no options available at current index
    backtrack = True
  else:
    backtrack = False
    next_codon = options[prev_choices[i]]
    prev_choices[i] += 1
    if i<5: #for first 5 codons, don't check for repeats or GC%
      nt_sequence = _append_codon(nt_sequence, next_codon, i)
      prev_codon = next_codon
      tot_gc += next_codon.count("G") + next_codon.count("C")
```

```
        i += 1
      else: #check for longer repeats or GC% for rest
        nt_sequence = _append_codon(nt_sequence, next_codon, i)
        next_gc = next_codon.count("G") + next_codon.count("C")
        tot_gc += next_gc
        gc_range = gc_perc*(i/(hex_index+0.0))
        gc_check = tot_gc/(i*codon_l+0.0)
        if (gc_check > gc_range) and (gc_check < (1-gc_range)):
          # GC in range
          text = ''.join(nt_sequence[:((i-2)*codon_l)])
          pattern =
              ''.join(nt_sequence[((i-2)*codon_l):((i+1)*codon_l)])
          if not pattern in text:
            # no repeating sequences, move on to next index
            prev_codon = next_codon
            i += 1
          else:
            # found repeating sequence, try again
            tot_gc -= next_gc
            if prev_choices[i] == len(options):
              backtrack = True
        else: # GC out of range, try again
          tot_gc -= next_gc
          if prev_choices[i] == len(options):
            backtrack = True
```

```
if backtrack == True:
    # all options have been tried at current index, go back one
    tot_gc -= prev_codon.count("G") + prev_codon.count("C")
    num_backtrack += 1
    prev_choices[i] = 0
    i -= 1
    prev_codon = nt_sequence[((i*codon_l)-3):(i*codon_l)]
```

Some of the other published algorithms account for only one biological constraint in their mapping scheme, if any. This one constraint is to exclude repeating sequences of nucleotides, and most mappings do not allow repeats of 3 or 4 in a row [13, 19, 20, 9]. Additional biological constraints are handled elsewhere in the encoding algorithm, if at all.

Some of the encoding algorithms use a specific mapping scheme that is intertwined with the rest of the algorithm. Others separate the mapping from the rest of the encoding algorithm. This provides the option to swap out different mapping schemes from different algorithms. In the following sections, one of these types of algorithms, the Fountain Codes algorithm, is described. Then we discuss how the mapping scheme and translation process is integrated into that encoding algorithm.

4.2 Fountain Codes Analysis

Fountain Codes cover a class of erasure codes that are rateless and universal. The term rateless describes the idea that the number of packets sent over an erasure channel, from a sender to the receiver, is potentially limitless. The number of packets needed

to uncover the original message can be determined dynamically. Fountain Codes are also universal because a given number of packets can be generated to decode the message, regardless of the statistics of errors produced within the channel [27]. In other words, it is not required to predetermine the rate of the channel in order to calculate the optimal parameters of the erasure code.

Byers and his team pioneered the idea of Fountain Codes with Tornado Codes [11]. The basic idea is that an input data source is split up into K segments where each segment is l bits long. Encoded segments become packets known as droplets. Each encoded droplet is generated by combining multiple segments from the original data source. Those segments are combined with the exclusive-or (XOR) operation. The droplets are then sent over an erasure channel in what can be described metaphorically as a fountain. On the receiving side, the decoder must collect enough droplets in order to uncover the transmitted message.

The number of segments per packet is small compared to the total number of segments in the data source. The collected droplets then form a sparse bipartite graph, where each packet is a vertex with edges connecting it to each included segment. Since this graph representation is so sparse, the algorithm allows for extremely fast encoding and decoding [11]. More in depth descriptions of the encoding and decoding processes are explained in the next section and Figures 4.1 and 4.2 shows visualizations of the encoding and decoding, respectively.

4.2.1 The LT Code

The first practical implementation of Fountain Codes was developed by Michael Luby and are now known as Luby Transform (LT) codes [25]. LT codes use an enhanced

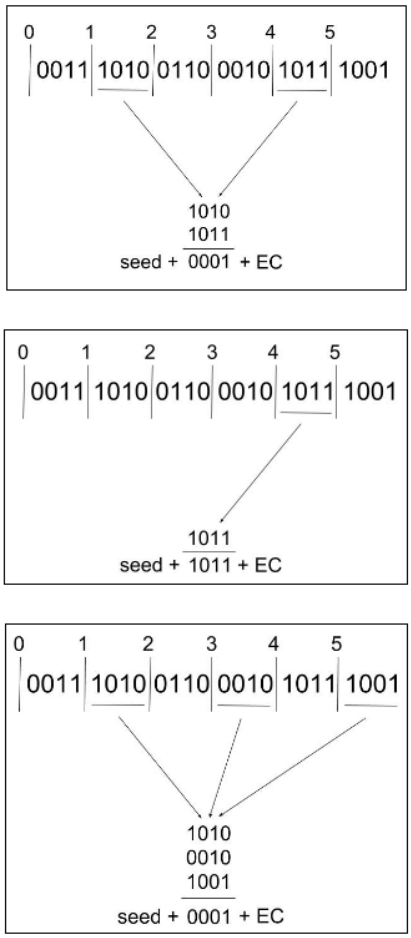


Figure 4.1: Visualization Example of Fountain Codes Encoding Process

version of the Soliton distribution for encoding and a message passing algorithm for decoding.

On the encoding side, the first step is to randomly choose a number, d_n, from a robust Soliton distribution. This distribution is described in detail in the next section. The value, d_n, represents the number of segments that will be contained in a given droplet, or the droplet's degree. Next, a d_n number of segments are chosen randomly, with replacement, from the original data source. Each segment is chosen

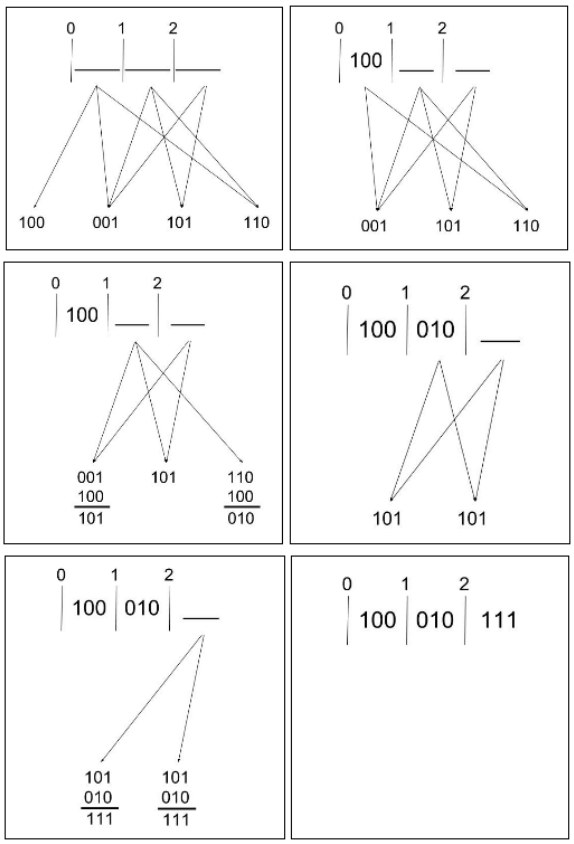


Figure 4.2: Visualization Example of Fountain Codes Decoding Process

over a uniform distribution. A key value is used to seed each of the two random number generators and is sent as header information with each droplet. The rest of the information in the droplet is the combined segments, where the XOR operation is used over the segments.

The idea for the decoding algorithm is to start with single-segment droplets and propagate that information through the other droplets until all segments are recovered. For any given droplet received, the key value is once again used to determine d_n and the d_n chosen segments. At this point, if there are multiple segments contained in a given droplet, then the original values of the segments are not yet

known. When a droplet is received that contains only one segment, the payload value is set to the corresponding segment and that segment is marked as decoded. Then, for all other droplets that also contain that same segment, the value is removed from those droplets. The XOR operation is used on the decoded segment and the corresponding droplets to effectively remove that segment from the droplet. This process continues and eventually, some droplets will be left with one segment. Again, that segment can be marked as encoded, the value saved in the final message, and propagated throughout the other droplets. These actions are repeated until all segments have been recovered from droplets and the message is decoded.

Degree Distribution

The degree distribution is an integral part of the LT code design. Since random number generators are employed in this algorithm, there is a small chance that the decoder will be unable to recover the encoded message. The degree distribution can be tuned to give the algorithm a higher probability of successfully decoding the message. More specifically, there are two main requirements that the degree distribution should accomplish. The first is that the encoder should generate enough single-segment packets to start off the decoding process. Secondly, each segment needs to be included in at least one droplet. These values are dependent on the size of the translated data set. Although, only one degree-one droplet would be required to start the decoding process. Then, after each iteration of processing the degree-one droplet, another single-segment would be uncovered, and so on. Ideally, the Soliton distribution would generate these steps. This distribution is represented by equation 4.1, where K is the total number of segments in the original data and d is the degree of a given droplet.

$$\begin{aligned} \rho(1) &= 1/K \\ \rho(d) &= \frac{1}{d(d-1)} \quad \text{for } d = 2, 3, \ldots, K \end{aligned} \tag{4.1}$$

Unfortunately, this works poorly in practice [27]. This distribution can produce situations where there are not enough, if any, degree-one droplets to start or continue the decoding operations. This is especially true if there are errors in the erasure channel and droplets can be lost. There is also the chance that some segments from the dataset are not included in any droplets at all.

Including two more parameters in the distribution allows a user to set the expected number of degree-one droplets over the entire process. The expected number of degree-one droplets is represented by S in equation 4.2 and the new parameters are δ and c.

$$S \equiv c \log_e(K/\delta)\sqrt{K} \tag{4.2}$$

A new function, $\tau(d)$, is used to augment the Soliton distribution is shown in equation 4.3. This function is based off of the expected value S.

$$\tau(d) = \begin{cases} \frac{s}{K}\frac{1}{d} & \text{for } d = 2, 3, \ldots, (K/S) - 1 \\ \frac{s}{K} log(S/\delta) & \text{for } d = K/S \\ 0 & \text{for } d > K/S \end{cases} \tag{4.3}$$

Finally, equation 4.4, shows the robust Soliton distribution, $\mu(d)$. This is produced by adding the ideal Soliton distribution ρ to the augmented distribution τ, and normalizing.

$$\mu(d) = \frac{\rho(d) + \tau(d)}{\sum_d \rho(d) + \tau(d)} \tag{4.4}$$

The graph in Figure 4.3 shows what the two distributions look like for a given case. We can see that at $d = 1$, there is a larger probability, so there is a better chance of generating degree-one droplets. These degree-one droplets ensure the progression of the decoding process. There is also a "spike" in the graph at the value K/S, which is about 41 for the given case. This spike helps to ensure that every segment from the data set is included in at least one droplet. This is the equation that we used for our Fountain Codes implementation and the parameters δ and c can be set by the user.

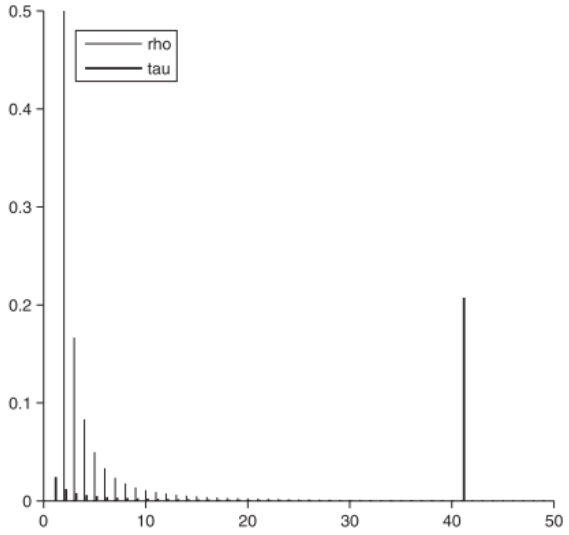


Figure 4.3: Robust Soliton Distribution [27] for the Case K=10000, c=0.2, δ=0.05

Delta and C Parameters

The choice of values for *delta* and c can present some trade-offs for the user to decide which features are more important. Both parameters represent values between zero

and one. There is less leeway offered for *delta* and is an easier decision than the user must make for c.

The *delta* parameter represents the probability of being unable to decode a message given that the decoder recovered an optimal number of droplets. We want the highest possible chance of discovering the message. Therefore, we choose *delta* to be very small, such as 0.001.

The c distribution parameter can be tuned to around 0.025. For a larger value of c, there are more single-segment droplets produced by the distribution. However, large values of c also decrease the degree value of the spike (K/S) and increase the number of droplets that have a degree of K/S. So there would be more droplets with a smaller degree. On the other hand, small values of c push the degree value of the spike to be larger. There would be fewer droplets with a large degrees, which in turn increases the average number of segments per droplet. Since all segments in a droplet must be removed to recover a final segment, if there are more segments within a droplet on average, then it would take longer for the decoding process. This translates to a longer computational runtime. Yet, small values of c also decrease the average number of recovered droplets needed to successfully decode the message. This represents a trade off between requiring less droplets overall to decode the message and an increase in running time.

We performed an analysis on the affects of the degree distribution based on varying levels of the c parameter. The analysis was conducted using python 2 in a Jupyter Notebook. For the datasets, we examined the degree per droplet produced from the robust Soliton distribution. The code used to produce this data was the DNA Fountain implementation written by Erlich and Zielinski. Since DNA Fountain includes a screening stage, we removed this to only examine the first 17,601 droplets

generated. This value of 17,601 droplets is the number of droplets needed to encode a 500 MB randomly generated binary file with 10% redundancy. We set each sequence to include 32 bytes for a total of 16,000 segments. Figures 4.4, 4.5, and 4.6 show the degree distributions for $c = 0.025$, $c = 0.1$, and $c = 0.5$.

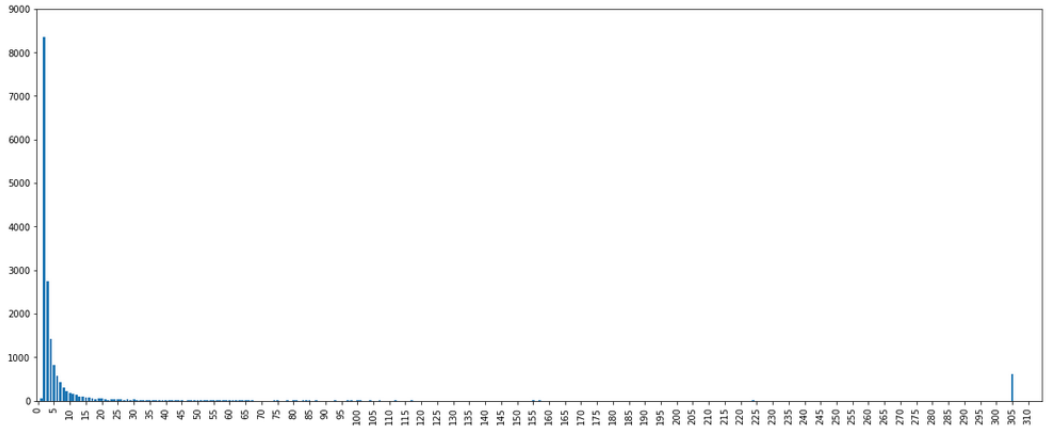


Figure 4.4: Robust Soliton Distribution for the Case K=16000, c=0.025, δ=0.001

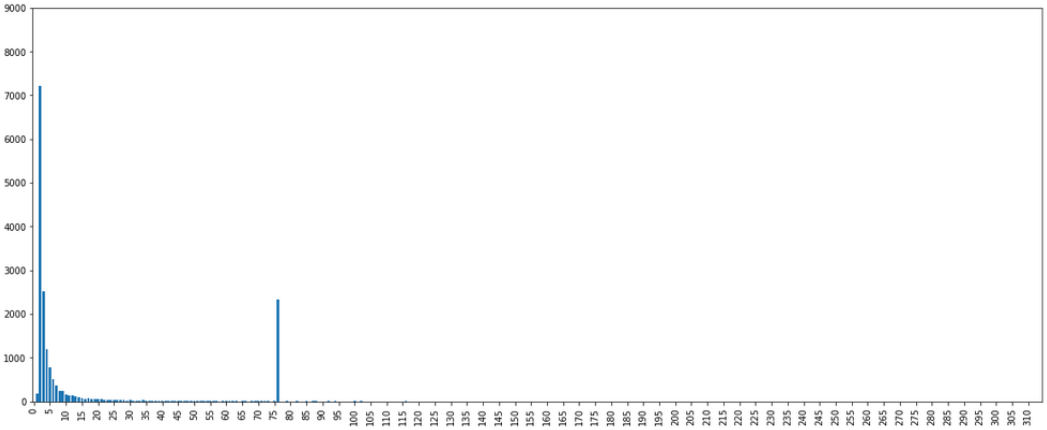


Figure 4.5: Robust Soliton Distribution for the Case K=16000, c=0.1, δ=0.001

Clearly, decoding a message is of utmost importance. However, the algorithm must be able to recover the message in a given amount of time. Depending on the

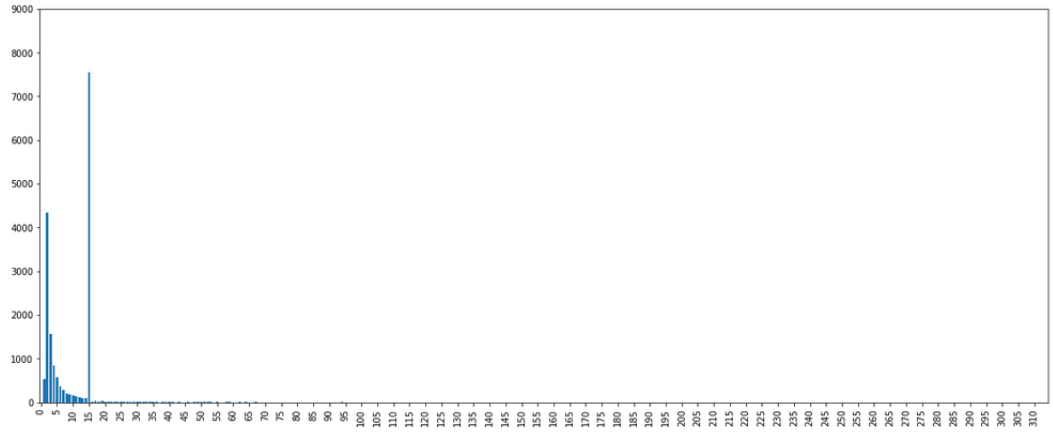


Figure 4.6: Robust Soliton Distribution for the Case K=16000, c=0.5, δ=0.001

situation, and the size of the files being processed, the user may not have time to wait for a file to be decoded. This equates to the situation of not being able to decode the message at all. Therefore, the user must determine appropriate values for *delta* and *c*. For our tests $\delta = 0.001$ and $c = 0.025$ proved to be successful choices.

4.2.2 DNA Fountain

In an application called DNA Fountain, Erlich and Zielinski implemented the Fountain Codes LT code for their algorithm for DNA data storage [17]. DNA Fountain contains a few adjustments so that the Fountain Codes algorithm can work specifically for DNA data storage. These alterations include a preprocessing stage and a post-processing stage. An overview of their algorithm is included in Section 3.7.

In the preprocessing stage, the data files to be stored are compressed. Not only does this step make the dataset smaller, it also gives the data a large amount of entropy, or randomness. The randomization of data is important for the adjustments made in the post-processing stage. The type of compression tool used can be deter-

mined by the user, however, Erlich and Zielinski used gzip in their experiments. After compressing the dataset, the next steps are the same as Fountain Codes. Based on the packet length l, chosen by the user, the dataset is split up into K equal length segments. The robust Soliton distribution is used to generate droplets based off of the standard LT codes algorithm and the seed used for the PRNGs is prepended to each resulting sequence.

The post-processing stage includes the possible addition of an error correction code, converting the data with a mapping scheme, and screening droplets based off of biological constraints. For the error correction code, DNA Fountain uses a Reed Solomon code that is appended to each droplet's sequence (for an in depth description of Reed Solomon codes, see Section 2.3). This can be checked for errors during decoding. Next, bits are converted to nucleotides with a basic mapping scheme, shown in Table 4.4. After each droplet's sequence is converted to nucleotides, the droplet reaches the screening stage.

Table 4.4: Basic Mapping Scheme Used by DNA Fountain

bits	nucleotide
00	A
01	C
10	G
11	T

The screening stage checks that each droplet's sequence conforms to two of the biological constraints discussed in Section 2.2. The algorithm watches for sequences that contain repeating nucleotides or an amount of GC content that is outside of an accepted range around 50%. If the sequence of a given droplet is found to have either of these two properties, the droplet is thrown out. The specifics of these constraints can be set by the user, however, the DNA Fountain implementation was tested with

three for the maximum amount of repeating nucleotides and a range of 5% around 50% GC content. While many droplets are screened in this stage, Fountain Codes are rateless, and the algorithm can continue to generate droplets. When droplets are excluded for not passing both biological constraints, more droplets are simply generated until a certain number of sequences are accepted. The number of sequences generated can be determined by the user for a variable amount of redundancy.

At this point, it is important to note that the Fountain Code is no longer rateless. Once a given number of sequences are generated, those are the only ones that will be stored. When decoding the information from the DNA, more droplets cannot be generated and the system must be able to recover the dataset from the stored DNA. Since information can also be lost during the synthesis, storage, and sequencing processes, generating the number of droplets necessary for recovering the information is a vital feature of the algorithm. We performed an analysis on the DNA Fountain implementation in order to determine the overall ability to recover encoded datasets.

Known Limitations

It was discovered that the DNA Fountain implementation was not successful in decoding the encoded data in all cases. There are some small flaws in the DNA Fountain implementation of the Fountain Codes algorithm that can make the data recovery improbable.

In particular, the screening stage of DNA Fountain can have a negative impact on the performance of the algorithm in practice. In test runs for decoding random binary data of different sizes, the average number of screened droplets can reach around 90%. For more details on these tests, see Chapter 5. This means that there is a high probability of loosing any given droplet even before DNA synthesis, storage,

and sequencing. Since a certain distribution of degree values for each droplet is expected, and certain single-segment droplets are needed for the decoding process, this screening can potentially remove the metaphorical pieces needed to complete the puzzle.

Specifically, this algorithm does not work well for encoding small files (less than 100 KB) because it is difficult for the algorithm to achieve the desired degree distribution and to gather the needed droplets. For example, an analysis on a 1 KB file found that two of the single-segment droplets had the same segment. Fortunately, storing small files is not an efficient use of the DNA data storage architecture at this point in time.

This algorithm also works well if the dataset is compressed beforehand. After a compression algorithm is applied, the resulting randomized data is less likely to map to sequences that would be screened out. If the dataset is compressed and efficiently randomized, single-segment droplets are not likely to be screened out more often than other multiple-segment droplets. However, since there is a high probability that any given droplet will be excluded, there is potential to loose a highly important "linchpin" type of droplet that could make the difference in a successful decoding process. Erlich and Zielinski do not compress the data within their DNA Fountain program but do specify that the process should be previously executed.

In the DNA Fountain program a bug was discovered on the decoding side that could effect the ability of the program to decode an encoded dataset. As was previously described, the encoding screens sequences for homopolymer runs and GC content. Therefore, sequences that do not pass these requirements should not be seen when attempting to decode. If sequences are seen on the decoding side that would not have passed screening during encoding, then there must have been an error

that occurred in that sequence and the program would not process those sequences. Screening for these types of sequences on the decoding side is a valid step but the code is located in the wrong spot. On the encoding side, this stage happens after the seed and error correction values are appended to the payload. However, on the decoding side, the screening happens after the error correction value has been removed. Because of this difference (with and without the error correction value), the GC content may be different and the decoding ends up excluding sequences that did not have errors in them. Excluding these extra sequences can have a negative impact on the program's ability to recover the original information.

Another issue with the DNA Fountain program is the runtime performance. For files larger than 10 MB, the decoding time becomes an issue for a realistic DNA data storage application. Since a DNA data storage architecture would be needed to store large amounts of data, an application that takes days or even weeks to decode tens to hundreds of megabytes of information is not feasible. A decrease in runtime was found in two main areas. The first is the screening stage. It takes the encoding longer by generating more droplets in order to find a given number of valid sequences that can pass the screening stage. Secondly, a bug was discovered in the decoding code that added an exponential amount of processing time. This single line of code mapped the decoded bytes of data to a char representation before writing to the output file. This line is not necessary as python allows bytes to be written directly to a file.

Our implementation attempts to improve upon these issues and is described in the next section. The results of our implementation compared to the DNA Fountain program is found in Chapter 5. Other issues were found to be outside the scope of this research and are described in Section 6.2.

4.3 Integration & Updates

For our encoding and decoding implementation, we chose to integrate our hex-to-codon mapping scheme and translation process into the DNA Fountain codes algorithm. After completing a detailed survey of other works (found in Chapter 3), we decided that the Fountain Codes algorithm was one of the most promising designs. The Fountain Codes algorithm balances a variable amount of parity and redundancy that can be fine-tuned by the user. Another helpful element of this design is the use of the XOR operation which can minimize repeating patterns within the data. Since the indexing seed values represent droplets instead of sequences, this algorithm can be configured to be scalable for large amounts of data. The Fountain Codes algorithm is also ideal because the mapping scheme is completely separate from the rest of the encoding and decoding processes. This allows us to easily swap out the mapping that Erlich and Zielinski used for our own which takes more biological constraints into account.

Much of the algorithm follows the original LT codes design for Fountain Codes. We split the data into equal length segments, where the number of bytes encoded in each segment is determined by the user. From those segments, we generate droplets with the same robust Soliton distribution. We also use a Reed Solomon code appended to each droplet's sequence. Then, the major difference is that we convert each droplet's sequence of binary data to a sequence of nucleotides using the mapping scheme and translation process detailed in Section 4.1. On the decoding side, the map is also used to convert nucleotides back to their binary representation, before the segment recovery process can begin. More details about the code implementation can be found in the documentation located in Appendix A.

Besides employing our hex-to-codon mapping scheme, we integrated two other major updates to the algorithm. These updates include removing the screening stage and altering the decoding algorithm.

4.3.1 Removed Screening Stage

Unlike Erlich and Zielinski's DNA Fountain implementation, we did not include a screening stage. Since the hex-to-codon mapping scheme and translation process accounted for the biological constraints that the screening process looked for, that step was no longer needed. Now, for our implementation, droplets are no longer discarded if their sequences do not pass the two biological constraints. Instead, our translation process attempts to find a valid sequence within a given number of backtracks. By setting a limit for the number of times the system can backtrack, we shorten the running time. However, there is a chance that the system cannot find a valid sequence within the limit of times backtracked. If a valid sequence is not found, the droplets with these types of sequences are then excluded. Although, by moving the biological constraint satisfaction process from a post-processing screening stage to the translation process we still observed two main improvements over the DNA Fountain algorithm.

The fact that droplets are not excluded at as high of a rate has a positive affect on the the algorithm's ability to recover encoded data. Our algorithm typically discards about 10% of the droplets generated, while DNA Fountain can exclude up to 88%. Since our algorithm does not exclude a majority of droplets, we are more likely to reach the intended distribution. We are also more likely to keep the single-segment droplets that are imperative to the successful recovery of the data. As an unintended side affect, we also achieved an decrease in running time. Since DNA Fountain has

to generate more droplets than are actually accepted, there is a waste in processing time. Another chance to improve the running time was discovered in the decoding algorithm.

4.3.2 Altered Decoding Algorithm

We made changes to the decoding algorithm in order to speed up the computational running time. More specifically, we changed the way single-segment droplets are propagated throughout other droplets with the same segment. Erlich and Zeilinski's DNA Fountain implementation used a depth-first approach for following the path of connected droplets. We instead used a breadth-first approach which showed a significant decrease in running time.

After the preliminary recovery and translation of each sequence, the key value is used to seed both random number generators and eventually, each droplet knows how many segments it contains and exactly which segments from the original dataset it contains. Similar to the DNA Fountain decoding algorithm, our implementation also starts by examining droplets with exactly one segment. Now, other recovered droplets may also include that segment and need to remove it. The difference in our decoding approach is the order in which individual recovered segments are removed from the other droplets that also contain that segment.

DNA Fountain solves the depth-first approach with a recursive function. The function starts with a recovered segment. It then finds other droplets that have the same segment. It starts at the beginning of that list of other droplets and removes the segment from the first droplet. There is a chance that the first droplet in the list only has two total segments. So if the initial segment is removed, the process has now recovered one other single-segment. Instead of moving on to the other droplets that

contain the initial segment, the function now processes the newly recovered segment. Each time a single-segment is recovered, it is processed immediately.

On the other hand, we implemented a breadth-first approach for processing these single-segments. Therefore, when a recovered segment is removed from a droplet with two segments and a new single-segment is discovered, that segment is added to a first-in-first-out (FIFO) queue, instead of being immediately processed. The starting single-segment is first removed from all other droplets that also contain that segment. Any time a new single-segment is discovered, it is simply added to the queue. Once the starting single-segment is finished being removed from all other droplets, the algorithm moves on to process the first segment in the queue. This process is repeated until the queue is empty. If all single-segments in the queue have been processed and the dataset has not been completely encoded, more droplets are processed.

The results of a breadth-first decoding approach versus a depth-first approach includes less calls to the XOR operation in the code. For example, for tests run on a randomly generated one megabyte file, the depth-first DNA Fountain implementation made 723,706 calls to the XOR operation. On the other hand, our REDNAM breadth-first implementation only made 14,636 calls to the XOR operation. The line of code for the XOR operation is one of the most expensive in terms of computational runtime. We believe that the reduction in the number of XOR operations has significantly reduced the computational runtime for the overall decoding program. Besides changing this algorithm, other small changes were made to clean up the code and decrease the running time. Chapter 5 details the results of our runtime tests which showed a speedup of 99.77 percent for decoding on a randomly generated 50 MB file.

CHAPTER 5

EVALUATION

In order for a DNA data storage architecture to function well in practice, it requires an encoding and decoding algorithm that can successfully recover the stored information without error. Performing lab tests in conjunction with REDNAM is outside the scope of this project. Due to high cost and time constraints, conducting experiments in synthesizing and sequencing the data produced by REDNAM is not feasible at this time. Instead of performing lab tests, our algorithm has been tested by running the encoding and decoding programs on various digital data files with various parameters. To test REDNAM's ability to recover from errors in the DNA, the sequences produced from the encoding program are artificially perturbed with random insertion, deletion and mutation errors. These tests show that REDNAM can recover encoded digital data when insertion, deletion and mutation errors occur in the DNA sequences. Our tests also reveal a speed up in the running time of our algorithm in comparison with Erlich and Zielinski's DNA Fountain implementation that is up to four hundred times faster on larger files.

All tests were run on a computer running Ubuntu 17.10 with 64 GB memory and an Intel Core i7-6700 CPU at 3.40 GHz.

5.1 Validation Testing

A python script was created to artificially alter the encoded data. This program mimics errors that may occur during the synthesis, storage or sequencing processes for writing and reading the DNA. Corruption of the data that may happen during any of these three stages can include insertions, deletions or mutations of nucleotides, as well as lost sections of sequences or loss of entire sequences.

The program, called degrade.py, starts by reading in a file that contains a list of nucleotide sequences. It is assumed that this file is generated by an encoding program. Next, the program randomly produces insertion, deletion and mutation errors within the nucleotide sequences. Lastly, the corrupted sequences are written to an output file that can be run in a decoding program.

Errors on a nucleotide are introduced into the sequences randomly, where the probability for insertions, deletions, and mutations are provided by the user. There is also a chance that entire sequences could be lost. More specifically, the main degrade code includes a for loop that randomly draws sequences from the input file. This loops runs for five times the number of input sequences. Then, for each drawn sequence, another loop runs three times. For each time in the inner loop, there are three conditional statements. The first statement randomly determines if an insertion should be made, based off of the probability given by the user. The second statement is a deletion, and the third statement is a mutation. If the execution enters one conditional, it will pass on the other two. Therefore, there can only be up to three errors per sequence. If an insertion error is selected, a new nucleotide to add is randomly chosen out of the four options and added somewhere randomly in the sequence. If a mutation error is selected, a nucleotide is randomly chosen from

the sequence and replaced with a new one. Finally, if a deletion error is selected, a nucleotide is randomly chosen from the sequence and removed.

The python code below shows the method that alters sequences.

```
def degrade(sequences_in, mutate, insert, delete):

    ins_prob = int(insert*10)
    del_prob = int(delete*10)
    mut_prob = int(mutate*10)

    errors = [0,0,0,0]
    nts = ["A", "T", "C", "G"]
    sequences_out = []
    times = len(sequences_in)*5

    for i in xrange(times):
        drawn_seq = random.choice(sequences_in)

        for j in xrange(3):

            ins_chance = random.randint(0,100)
            if ins_chance < ins_prob:
                errors[0] += 1
                errors[1] += 1
                pos = random.randint(0,len(drawn_seq)-1)
```

```
            nt_new = random.choice(nts)
            drawn_seq = drawn_seq[:pos] + nt_new + drawn_seq[pos:]
            continue

        del_chance = random.randint(0,100)
        if del_chance < del_prob:
            errors[0] += 1
            errors[2] += 1
            pos = random.randint(0,len(drawn_seq)-1)
            drawn_seq = drawn_seq[:pos] + drawn_seq[(pos+1):]
            continue

        mut_chance = random.randint(0,100)
        if mut_chance < mut_prob:
            errors[0] += 1
            errors[3] += 1
            pos = random.randint(0,len(drawn_seq)-1)
            nts.remove(drawn_seq[pos])
            nt_new = random.choice(nts)
            nts = ["A", "T", "C", "G"]

        sequences_out.append(drawn_seq)

    return sequences_out, errors
```

Tables 5.1, 5.2, and 5.3 show different levels of degradation on different file sizes. The levels of degradation are none, light, medium, and heavy, where there is a 0, 0.25, 0.5, or 0.75 chance of insertions, deletions and mutations for each level. We also alter the number of bytes used for the Reed Solomon error correction code and the level of redundancy. These tables demonstrate REDNAM's ability to recover the original digital data after the nucleotide sequences have been corrupted with errors.

Table 5.1: Degradation Tests - 2 Bytes Reed Solomon, 10% Redundancy

		none - 0	light - 0.25	medium - 0.5	heavy - 0.75
100 KB	encode time	0m2.060s	0m2.019s	0m2.038s	0m2.039s
	decode time	0m0.659s	0m1.566s	0m1.360s	0m1.240s
	insertions	0	1018	2642	3684
	deletions	0	1055	2484	3388
	mutations	0	1013	2420	3137
	success	yes	no	no	no
500 KB	encode time	0m9.400s	0m9.288s	0m9.383s	0m9.449s
	decode time	0m2.755s	0m6.480s	0m6.101s	0m5.483s
	insertions	0	5305	12985	18444
	deletions	0	5098	12591	17022
	mutations	0	5076	11793	15805
	success	yes	yes	no	no
1 MB	encode time	0m18.826s	0m18.768s	0m18.908s	18.998s
	decode time	0m5.502s	0m12.150s	0m12.218s	0m11.137s
	insertions	0	10718	26744	37491
	deletions	0	10648	25848	34978
	mutations	0	10170	23929	32412
	success	yes	yes	no	no
10 MB	encode time	3m14.927s	3m12.930s	3m16.171s	3m16.979s
	decode time	1m5.413s	1m56.326s	2m5.378s	2m12.497s
	insertions	0	107028	267818	376232
	deletions	0	104693	253484	348990
	mutations	0	102766	241460	324671
	success	yes	yes	yes	yes

Table 5.2: Degradation Tests - 10 Bytes Reed Solomon, 10% Redundancy

		none - 0	light - 0.25	medium - 0.5	heavy - 0.75
100 KB	encode time	0m2.863s	0m2.850s	0m2.874s	0m2.852s
	decode time	0m1.134s	0m3.589s	0m3.001s	0m2.725s
	insertions	0	1009	2572	3802
	deletions	0	1067	2464	3311
	mutations	0	1001	2447	3215
	success	yes	no	no	no
500 KB	encode time	0m13.754s	0m13.699s	0m13.859s	0m13.304s
	decode time	0m5.159s	0m14.793s	0m14.751s	0m13.161s
	insertions	0	5245	12847	18126
	deletions	0	5076	12547	16932
	mutations	0	5194	11849	16224
	success	yes	yes	no	no
1 MB	encode time	0m27.257s	0m26.710s	0m26.760s	0m26.896s
	decode time	0m10.043s	0m26.772s	0m29.792s	0m26.720s
	insertions	0	10751	26721	37433
	deletions	0	10506	25474	34744
	mutations	0	10357	24549	32418
	success	yes	yes	yes	no
10 MB	encode time	4m31.004s	4m30.642s	4m33.947s	4m32.265s
	decode time	1m49.619s	4m10.141s	4m27.049s	4m46.320s
	insertions	0	106779	268062	375308
	deletions	0	105018	253058	349562
	mutations	0	103065	241388	324409
	success	yes	yes	yes	yes

The results of Table 5.1 shows how REDNAM reacts to varying levels of errors with default parameters for two bytes of Reed Solomon error correction code and ten percent redundancy. In general, smaller files are less likely to be able to recover the original data. Larger files are more robust because they are more likely to have a degree distribution that matches the robust Soliton distribution of the LT code. These degradation tests also show that the computational runtime of the decoding program takes longer to complete when errors are introduced. This result is expected

Table 5.3: Degradation Tests - 2 Bytes Reed Solomon, 20% Redundancy

		none - 0	light - 0.25	medium - 0.5	heavy - 0.75
100 KB	encode time	0m2.255s	0m2.264s	0m2.286s	0m2.229s
	decode time	0m0.691s	0m1.207s	0m1.405s	0m1.326s
	insertions	0	1113	2842	3944
	deletions	0	1149	2757	3708
	mutations	0	1094	2674	3603
	success	yes	yes	yes	yes
500 KB	encode time	0m10.336s	0m10.156s	0m10.027s	0m10.273s
	decode time	0m2.806s	0m5.170s	0m5.516s	0m5.773s
	insertions	0	5663	14176	19964
	deletions	0	5610	13421	18265
	mutations	0	5432	12683	17203
	success	yes	yes	yes	yes
1 MB	encode time	0m20.708s	0m20.607s	0m20.806s	0m20.364s
	decode time	0m5.465s	0m9.760s	0m10.505s	0m11.007s
	insertions	0	11550	29557	40677
	deletions	0	11551	27710	37720
	mutations	0	11168	26330	35484
	success	yes	yes	yes	yes
10 MB	encode time	3m32.207s	3m32.226s	3m33.201s	3m31.334s
	decode time	1m5.722s	1m40.367s	1m49.603s	1m55.403s
	insertions	0	116749	292728	408659
	deletions	0	114455	277405	380463
	mutations	0	111266	263476	354195
	success	yes	yes	no*	yes

*Finished decoding but incorrect value.

because when some sequences are dropped for errors, it will take longer to process more sequences.

Table 5.2 shows the results of different levels of degradation with an increased number of bytes of the Reed Solomon code. This increase in the size of the Reed Solomon code did not show much improvement over the two bytes. The only difference was in the successful recovery of the 1 MB file at the medium degradation level. Overall, calculating the Reed Solomon code during encoding and validating the code

during decoding added an increase in the computational runtime of both the encoding and decoding programs.

The last table, Table 5.3, shows a ten percent increase in the level of redundancy for a total of twenty percent. When compared with the other two degradation tables, these tests show the ability to recover all but one of the files over the three levels of degradation. The only file that was not successfully recovered was the 10 MB file at medium redundancy. This test was able to complete the decoding process, however there was a difference in bytes in the recovered file when compared with the original.

In an attempt to find the error that caused the file to be corrupt, we were able to determine which droplet was faulty. Without keeping track of the original droplet's sequence all the way through to the error, it is difficult to say with certainty exactly what type of error caused the recovery process to break. However, a comparison of the sequence that was degraded with what the correct sequence likely should have been, it seems there was an insertion error and a deletion error that shifted a sequence of nucleotides off by one base. Another issue could have been a mutation that occurred within the Reed Solomon error correction code which caused the code to incorrectly alter the data section of the sequence. Unfortunately, the erroneous droplet led to an incorrect single segment value that propagated throughout many other droplets and caused the decoded data to be corrupt. The effects of this situation and the possible ways to correct them can be researched further and a few of these ideas are presented in the Future Work section of Chapter 6.

Overall, the results of these degradation tests demonstrate that an increase in the percentage of redundancy will aid in the recovery process better than an increase in the number of Reed Solomon bytes. This result is not surprising because the decoding program is currently designed to discard sequences that are too long or too

short. Therefore, only mutation errors can be detected and corrected by the Reed Solomon code. Sequences with insertion or deletion errors are discarded so the level of redundancy can recover that lost information.

It should also be noted that the light, medium and heavy levels were chosen to demonstrate the robustness of the encoding and decoding programs. However, studies show that the actual chance of errors is likely smaller than our lightest level. Experiments by Organick, et. al. [29] show that the chance of errors on a nucleotide basis varies depending on the nucleotide type, its location in the sequence and the surrounding nucleotides. Organick, et. al. found the highest error rate was $1.5\text{x}10^{-3}$ for substitutions of T values. Ross et. al. [32] also performed an analysis of error rates in sequence data for different sequencing technologies.

In general, it is up to the user to determine a balance between the level of redundancy and the number of bytes of Reed Solomon codes needed. These two values can have an affect on the computational runtime of the encoding and decoding programs as well as the cost associated with synthesizing the length and the number of sequences generated. Based on experiments for error rates, degradation levels, and the results of these tables, the user should determine the parameters that they require for the successful recovery of their data when using REDNAM.

5.2 Running Time

As was discussed in Sections 3.7 and 4.2, Erlich and Zielinski produced encoding and decoding programs that implemented the Fountain Codes algorithm, called DNA Fountain. In order to see how REDNAM compared to DNA Fountain, the encoding and decoding programs were run on the same files with the same parameters. These

test files are listed below. Table 5.4 shows the running times for encoding, the running times for decoding, the ratio of accepted droplets out of the total number generated, and whether or not the test was successful in recovering the original file.

The main reason for these tests was to compare the computational runtime of the programs. Therefore, no errors were added to the sequences between the execution of the encoding and decoding programs. It should also be noted that the two bugs found in DNA Fountain, including the incorrect screening step and the extraneous mapping step were removed before conducting these tests. More details on these bugs can be found in Section 4.2.

Description of test files:

- 8 Randomly generated binary files ranging in size from 100 KB to 100 MB.
- Erlich and Zilinski's dataset [17], a 2.1 MB compressed file including:
  - The Kolibri operating system
  - The Arrival of a Train (Movie)
  - The Pioneer Plaque (Image)
  - Shannon's manuscript on information theory (PDF format)
  - $50 Amazon gift card (text)
  - An empty 4.5 PB compressed file (Malware)
- Artaméne or the Grand Cyrus (1649-1653) by Madeleine de Scudéry, the longest novel in French literature, an 11 MB text file [1].
- A compressed version of the aforementioned text file, 3 MB in size.

- Goldman's dataset [19], a 662 KB compressed file including:
  - All 154 of Shakespeare's sonnets (ASCII text)
  - Watson and Crick's paper on the molecular structure of nucleic acids (PDF format)
  - A medium-resolution color photograph of the European Bioinformatics Institute (JPEG 2000 format)
  - A 26 second excerpt from Martin Luther King's 1963 'I have a dream' speech (MP3 format)
  - A Huffman code (ASCII text)

Encoding Command for DNA Fountain:

```
$ time python encode.py \
--file_in $1 \
--size 32 \
-m 3 \
--gc 0.05 \
--rs 2 \
--delta 0.001 \
--c_dist 0.025 \
--alpha 0.1 \
--no_fasta \
--out $1.dna
```

Decoding Command for DNA Fountain:

```
$ time python receiver.py \
-f $1 \
--header_size 4 \
--rs 2 \
--delta 0.001 \
--c_dist 0.025 \
-n $2 \
-m 3 \
--gc 0.05 \
--max_hamming 0 \
--out $1.out
```

Encoding Command for our code:

```
$ time python encode.pyx \
-f $1 \
-l 32 \
--delta 0.001 \
--c_dist 0.025 \
--rs 2  \
--alpha 0.1 \
--out $1.dna \
--map original_map.txt
```

Decoding Command for our code:

```
$ time python decode.pyx \
-f $1 \
-n $2 \
-l 32 \
--delta 0.001 \
--c_dist 0.025 \
--rs 2 \
--out $1.out \
--map original_map.txt
```

Table 5.4 shows that there were two files which DNA Fountain was unable to recover the encoded file, including the 100 KB randomly generated file and the 11 MB text file. The 100 KB text file is likely too small for the screened Fountain Codes algorithm to achieve the needed degree distribution because tests on files smaller that 100 KB were also unsuccessful for both implementations. Secondly, DNA Fountain likely had trouble with the text file because it was not compressed. There may have been patterns in the textual data that could not be mapped to nucleotide sequences without passing the specified constraints in the screening stage. However, since our implementation can generate a larger ratio of valid sequences, we can better handle different types of data and possible patterns in a dataset.

REDNAM is most successful in the speedup of the computational runtime. Adjustments were made to optimize the code for both encoding and decoding programs. Although, the largest speedup comes from the different approach for the decoding algorithm, described in Section 4.3. The long run times for DNA Fountain does not

pose a problem for smaller files. Yet, for larger files, the time difference is significant. Tests completed on decoding a 50 MB randomly generated binary file took around two days to decode, while our implementation took only 6 minutes and 26 seconds, a speed up of over 400 hundred times as fast.

Table 5.4: Running Times for Encoding and Decoding Programs

	DNA Fountain					REDNAM			
	Encoding Time	Decoding Time	Accepted Droplet Ratio	Successfully Recovered File		Encoding Time	Decoding Time	Accepted Droplet Ratio	Successfully Recovered File
100 KB Random file	0m3.498s	0m0.803s	0.124	no		0m2.093s	0m0.694s	0.891	yes
500 KB Random file	0m17.143s	0m8.239s	0.126	yes		0m9.364s	0m2.826s	0.896	yes
1 MB Random file	0m37.285s	0m36.241s	0.125	yes		0m19.313s	0m5.437s	0.897	yes
2 MB Random file	1m15.924s	2m17.187s	0.125	yes		0m38.779s	0m11.631s	0.897	yes
3 MB Random file	1m55.886s	4m7.420s	0.124	yes		0m58.063s	0m17.576s	0.893	yes
10 MB Random file	6m42.105s	46m53.181s	0.124	yes		3m15.123s	1m5.425s	0.897	yes
50 KB Random file	35m13.860s	2748m7.409s	0.125	yes		16m48.657s	6m18.876s	0.897	yes
100 MB Random file	*	*	*	*		33m40.877s	13m11.811s	0.897	yes
2 MB Erlich .zip	1m16.383s	2m8.578s	0.125	yes		0m38.646s	0m11.793s	0.896	yes
11 MB Text file	7m57.105s	1m18.249s	0.113	no		4m30.223s	1m13.251s	0.842	yes
3 MB .zip text file	2m19.386s	17m44.188s	0.124	yes		1m8.149s	0m21.377s	0.897	yes
600 KB Goldman .zip	0m22.531s	0m11.856s	0.125	yes		0m12.020s	0m3.629s	0.896	yes

* Tests not completed due to long run times.

CHAPTER 6

CONCLUSION & FUTURE WORK

6.1 Conclusion

Using DNA as a data storage material shows promise despite the fact that this is a relatively new area of research. Initial iterations in developing encoding and decoding algorithms have shown some of the strengths and weaknesses in translating digital data to be stored in DNA. While recent algorithms have tried to balance information density with error correction and redundancy, not many have taken the idiosyncrasies of the material itself into account. In this thesis, we have explored an algorithm's ability to satisfy certain biological constraints while accounting for errors and accurately restoring encoded digital data.

We developed REDNAM, which includes a novel mapping scheme and translation process that can satisfy four different constraints that increase the chances of avoiding complications during the synthesis, storage and sequencing processes in the data storage architecture. This mapping scheme is inspired by nature and the translation process develops nucleotide sequences that conform to the following four constraints; excluding start codons, excluding more than three repeating nucleotides in a row, excluding repeating sequences longer than three codons in a given sequence, having between 45% and 55% GC content in the entire sequence. No other algorithm that we are aware of has accounted for all of these constraints in the generated sequences.

REDNAM also improves upon the adaptation of the Fountain Codes algorithm for generating DNA sequences by integrating our mapping scheme and translation process into the algorithm. Our translation process has a larger probability of generating a nucleotide sequence that follows the given set of constraints. This results in a program that is twice as fast at encoding because it does not need to generate as many droplets overall. The algorithm is also more likely to reach a robust Soliton degree distribution that matches the ones specified by the LT Code. Finally, changes made to the decoding algorithm also resulted in a speedup which makes the program more feasible for use in a DNA data storage product.

Different levels of redundancy can enhance the algorithm's ability to correct insertion, deletion and mutation errors that may occur. However, we have identified three different improvements that can be made to increase the robustness of this algorithm to errors while maintaining a low redundancy and high information density. These areas of future work are detailed in Section 6.2 and can help bring us one step closer to a viable algorithm that can be employed in a DNA data storage system.

6.2 Future Work

The Fountain Codes algorithm is a quality algorithm for sending data over information channels. We have shown that Fountain Codes can also be used for encoding and decoding information into DNA. However, Fountain Codes were originally intended to be rateless, which is not possible for the DNA data storage architecture. With this architecture in mind, adjustments can be made to the Fountain Codes algorithm in order for it to be more robust for a DNA data storage application. The Fountain Codes algorithm can be enhanced on the encoding side and decoding side. Error correction

can also be enhanced in order to account for insertion, deletion, and mutation errors that can occur on a per nucleotide basis during the DNA data storage process. These areas of further research are all outside the scope of this thesis and can be considered for future work.

6.2.1 Deterministic Segment Choices

One way to optimize Fountain Codes for use in DNA data storage is to use a more deterministic approach on the encoding side. The robust Soliton distribution is useful for randomly choosing which segments should be included in which packets. While the randomness is useful, the algorithm does not guarantee that the packets will include the segments needed to successfully decode the original information. This is especially an issue for small files, because there is a higher chance of selecting the same segment multiple times for a single-segment packets.

A simple way to determine that all segments are distributed throughout the packets correctly is to keep track of the packets during encoding. This way, the algorithm is essentially attempting to decode the packets as they are generated during the encoding stage. Then, the algorithm can stop generating packets once enough have been created to successfully decode them later.

Another way would be to deterministically select certain segments to go in certain packets while keeping track of the generated packets. This would be a better way to keep track of redundancy because it could keep track of redundant segments instead of generating an extra number of packets. This implementation could then keep track of the probability of successful decoding given that a certain number of packets were lost or compromised.

6.2.2 Enhanced Decoding

The decoding side of the Fountain Codes algorithm can be enhanced for better performance by augmenting the segment discovery stage. The intended implementation starts the segment discovery stage by first examining single-segment droplets. Then those single-segments are propagated throughout the other droplets with that same segment. This works well when the needed single-segments have been recovered successfully and start the avalanche of the decoding process. However, if the needed single-segment droplets are somehow lost then there is no way to continue on with the decoding process and the algorithm fails in decoding the original information.

Instead, the algorithm can examine pairs of packets with degrees that differ by one segment. It should attempt to find pairs of packets in which one packet contains all of the same droplets as the other packet with one extra segment. The XOR operation can be performed on the payload of the first packet and the payload of the second packet in order to reveal the value of the one segment that differed. Then this single-segment packet can be propagated throughout the other packets as usual.

This approach can be taken a step further for pairs of packets that differ by two degrees. The XOR operation can then be performed on the payload of the pair to reveal a value that includes two segments. This newly recovered value can potentially be combined with a packet with three segments in order to discover a single-segment value. The difficulty in this approach would be to design an implementation with optimal computational time.

This approach would also reduce the potential of having a "linchpin" packet that is needed for the decoding process to get going or to continue at any stage. Therefore, if this special packet was lost it would not compromise the ability of the decoding

algorithm to recover the information successfully. This would increase the robustness of the Fountain Codes algorithm overall.

6.2.3 Enhanced Error Correction

The Fountain Codes algorithm provides multiple areas to add different levels of error detection and error correction. For example, we augmented our approach with a Reed Solomon code on each packet. Besides an added error correction code, another step would be to use majority voting on redundant segments.

One of the features of Fountain Codes is that segments may be included in multiple packets. Therefore, each time a segment is revealed on the decoding side, the algorithm should keep track of it. When an individual segment is revealed multiple times, the algorithm could compare to make sure that the values match. If there is a discrepancy, majority voting could be used to decide which value was incorrect. Although, in implementing this approach, one would need to be careful as to not propagate a single segment value throughout other droplets if that segment is discovered to have an error. This could cause the error to propagate throughout multiple other segments.

If implemented correctly, this approach would have the added benefit of saving some of the information in a given droplet, even if an error has occurred. In the original Fountain Codes implementation, if an error is suspected in a given packet then it is thrown out. This could potentially discard useful information that is hiding underneath the error.

REFERENCES

[1] Madeleine et Georges de Scudery Artamene ou le Grand Cyrus.

[2] Magnetic Tape Storage and Handling, 1995.

[3] Sonly develops magnetic tape technology with the world's highest areal recording density of 148 Gb/inˆ2, 2014.

[4] Big Data Analytics, 2015.

[5] White Paper: Archival Disk Technology, 2015.

[6] 8-Bit AVR Microcontroller ATMega32A DATASHEET COMPLETE, 2016.

[7] Shilov Anton. Western Digital Announces Ultrastar He12 12 TB and 14 TB HDDs, 2016.

[8] Carter Bancroft, Timothy Bowler, Brian Bloom, and Catherine Taylor Clelland. Long-Term Storage of Information in DNA. *Science*, 293(5536):1763–1765, 2001.

[9] Meinolf Blawat, Klaus Gaedke, Ingo Hütter, Xiao-Ming Chen, Brian Turczyk, Samuel Inverso, Benjamin W Pruitt, and George M Church. Forward Error Correction for DNA Data Storage. *Procedia Computer Science*, 80:1011–1022, 2016.

[10] James Bornholt, Randolph Lopez, Douglas M Carmean, Luis Ceze, Georg Seelig, and Karin Strauss. A DNA-Based Archival Storage System. *ACM SIGOPS Operating Systems Review*, 50(2):637–649, 2016.

[11] John W Byers, Michael Luby, Mitzenmacher Michael, and Rege Ashutosh. Reliable to Distribution Approach of Bulk Data. *ACM SIGCOMM Computer Communication Review*, 28(4):56–67, 1998.

[12] Sam Chen. Samsung Unveils 32TB SSD Leveraging 4th Gen 64-Layer 3D V-NAND, 2016.

[13] George M. Church, Yuan Gao, and Sriram Kosuri. Next-generation digital information storage in DNA. *Science*, 2012.

[14] Catherine Taylor Clelland, Viviana Risca, and Carter Bancroft. Hiding messages in DNA microdots. *Nature*, 399(6736):533–534, 1999.

[15] Tom Coughlin. Keeping Data For A Long Time, 2014.

[16] Pierre Delforge. America's Data Centers Consuming and Wasting Growing Amounts of Energy, 2015.

[17] Yaniv Erlich and Dina Zielinski. DNA Fountain enables a robust and efficient storage architecture. *Science*, 355(6328):950–954, 2017.

[18] Kazuo Goda and Masaru Kitsuregawa. The history of storage systems. *Proceedings of the IEEE*, 100(100(Special Centennial Issue)):1433–1440, 2012.

[19] Nick Goldman, Paul Bertone, Siyuan Chen, Christophe Dessimoz, Emily M. Leproust, Botond Sipos, and Ewan Birney. Towards practical, high-capacity, low-maintenance information storage in synthesized DNA. *Nature*, 494(7435):77–80, 2013.

[20] Robert N. Grass, Reinhard Heckel, Michela Puddu, Daniela Paunescu, and Wendelin J. Stark. Robust chemical preservation of digital information on DNA in silica with error-correcting codes. *Angewandte Chemie - International Edition*, 54(8):2552–2555, 2015.

[21] Mark Haiman. Notes on Reed-Solomon Codes.

[22] Jonathan P.L. Cox. Long-term data storage in DNA. *TRENDS in Biotechnology*, 19(7):247–250, 2001.

[23] Michael Lesk. How Much Information Is There In The World?, 1997.

[24] Leo Leung. 99.8% of the world's data was created in the last two years., 2014.

[25] Michael Luby. LT codes. *Proceedings. The 43rd Annual IEEE Symposium on Foundations of Computer Science*, pages 271–280, 2002.

[26] David J C Mackay. Information Theory, Inference, and Learning Algorithms.

[27] D.J.C. MacKay. Fountain codes. *IEE Proceedings - Communications*, 152(6):1062 – 1068, 2005.

[28] Olgica Milenkovic, Ryan Gabrys, Han Mao Kiah, and S.M. Hossein Tabatabaei Yazdi. Exabytes in a Test Tube: The Case for DNA Data Storage, 2018.

[29] Lee Organick, Siena Dumas Ang, Yuan-Jyue Chen, Randolph Lopez, Sergey Yekhanin, Konstantin Makarychev, Miklos Z Racz, Govinda Kamath, Parikshit Gopalan, Bichlien Nguyen, Christopher Takahashi, Sharon Newman, Hsing-Yeh Parker, Cyrus Rashtchian, Kendall Stewart, Gagan Gupta, Robert Carlson, John Mulligan, Douglas Carmean, Georg Seelig, Luis Ceze, and Karin Strauss. Scaling up DNA data storage and random access retrieval. *bioRxiv*, 2017.

[30] Harlan Foote Pak Chung Wong, Kwong-Kwok Wong. Organic Data Memory Using the DNA Approach. *Communications of the ACM*, 46(1):95–98, 2003.

[31] I S Reed and G Solomon. Polynomial codes over certain finite fields. *Journal of the society for industrial and applied mathematics*, 8(2):300–304, 1960.

[32] Michael G. Ross, Carsten Russ, Maura Costello, Andrew Hollinger, Niall J. Lennon, Ryan Hegarty, Chad Nusbaum, and David B. Jaffe. Characterizing and measuring bias in sequence data. *Genome Biology*, 2013.

[33] SINTEF. Big Data, for better or worse: 90% of world's data generated over last two years, 2013.

[34] S. M.Hossein Tabatabaei Yazdi, Yongbo Yuan, Jian Ma, Huimin Zhao, and Olgica Milenkovic. A Rewritable, Random-Access DNA-Based Storage System. *Scientific Reports*, 5(14138), 2015.

[35] James Westall and James Martin. An Introduction to Galois Fields and Reed-Solomon Coding, 2010.

[36] Stephen B. Wicker. *Error Control Systems for Digital Communicatino and Storage*. Prentice-Hall, Inc., Upper Saddle River, New Jersey, 1995.

[37] Victor Zhirnov, Reza M Zadegan, Gurtej S Sandhu, George M Church, and William L Hughes. Nucleic acid memory. *Nature materials*, 15(4):366–370, 2016.

APPENDIX A

REDNAM CODE DOCUMENTATION

A program has been written in order to test and analyze the developed algorithm. This program is a command-line application written in Python 2 and compiled into Cython for extra speedup. Python was determined to be a good choice for three main reasons. One reason is the ease of readability for those without a background in computer science or for those with little experience in programming. Another reason for choosing python is that the website, using the flask framework, is also written in python. This allows the code to be reusable between the website and the standalone program. The third reason for using python is that some code was already available. Erlich and Zielinksi have their implementation of the fountain code algorithm available for download. Since our algorithm also uses part of the fountain code algorithm, we were able to make use of some of their code. Python also has a library implementation for Reed Solomon codes. This library can be imported into the program for use. Creating a program that reuses code and other libraries allows us to write less code and saves time during the development stage.

There are two main parts to this program. One part is the encoding algorithm that can take any file and convert it to a list of nucleotide sequences. The second part is the decoding algorithm. The decoding side can take a list of nucleotide sequences and convert them back to the original digital data file. There are ten python files that

include the code for encoding and decoding programs. Some files are only needed for the encoding program, some files are only needed for the decoding program, and some files are used by both. Below is a list of the python files created for these programs. Then, there is a description of the functionality that each file provides.

List of files:

- encode.pyx
- processing.pyx
- fountain.pyx
- droplet.pyx
- decode.pyx
- pool.pyx
- mapping.pyx
- transpose.pyx
- lfsr.pyx
- robust_soliton.pyx

Figure A.1 shows a UML diagram of the code used for the encoding and decoding programs and how they are connected. Appendix B shows the programs' usage and example output.

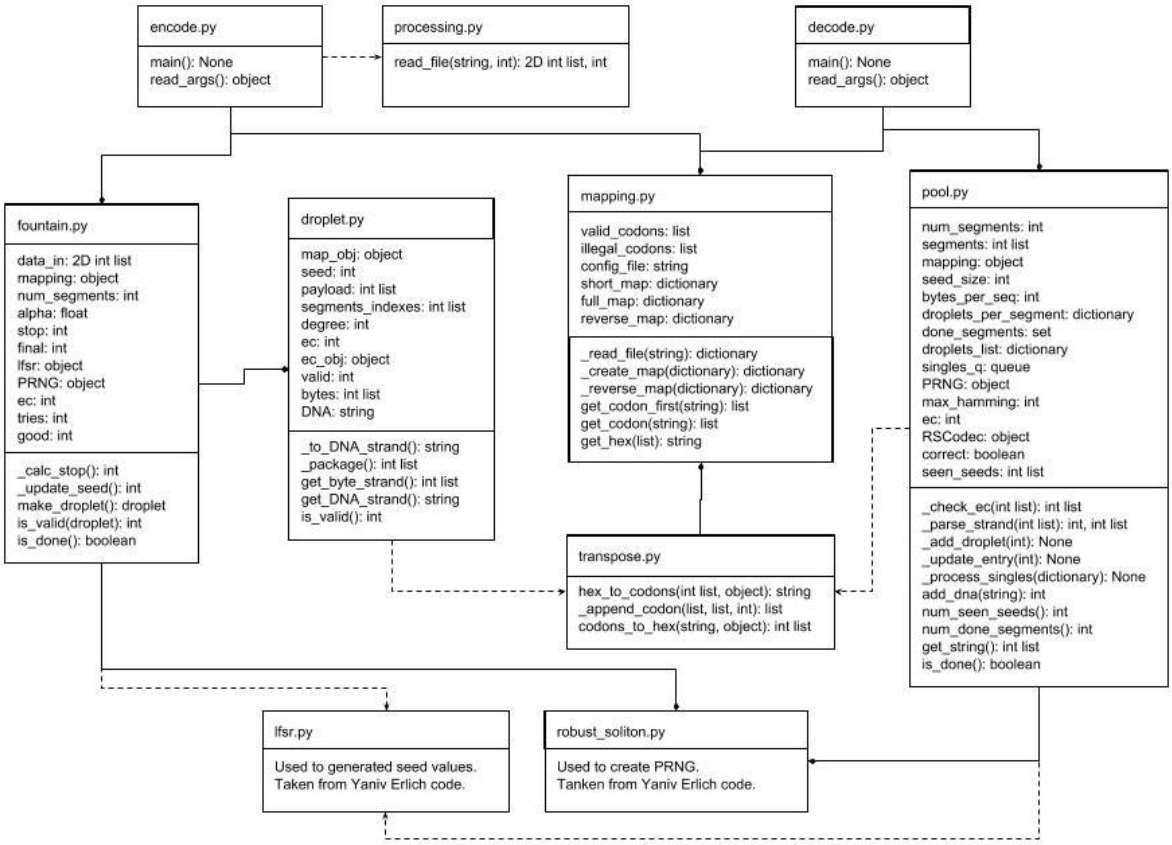


Figure A.1: UML Diagram of Encoding and Decoding Programs

encode.pyx

This file includes the main method for the encoding part of the program. It also includes a method for parsing command line arguments as well as setting up feedback for the user on program usage. After command line arguments are parsed, the functionality in the main method includes reading in the given file and setting up the fountain object. Then, a while loop is used to create droplet objects with a valid sequence, until enough are created. Valid droplet strands are written to the output file. Feedback is given to the user while the program is running in the form of logging messages and a progress bar.

processing.pyx

This file contains a helper method that reads in and prepares any digital data file for processing. It is used by the encoding portion of the program to load the file given by the user to be encoded. This method splits the entire file into segments of equal length, where the length in bytes is provided by the user. If the number of bytes in the file does not divide evenly by the number of bytes per segment, then the end of the file is padded with zeros. Finally, each byte of each segment is converted to its integer representation. The list of segments, where each segment is represented by a list of integers, is returned from this method for processing.

fountain.pyx

This file contains a majority of the functionality for the fountain codes encoding algorithm. It sets up the random number generators, creates droplets and checks them for valid sequences, all while keeping track of valid droplets in order to determine when no more droplets need to be created and the processing can stop. The main loop is controlled in the encode.pyx file, however the important code is contained in the fountain.pyx file. More specifically, the parameters needed for creating a DNAFountain object include data_in, mapping, delta, c_dist, ec, stop, alpha. The first parameter, data_in, represents the 2D list of integers that was previously created after reading in the given file to encode. The mapping parameter is associated with a Map_Object that represents the user provided mapping scheme. Delta and c_dist are both parameters needed for the random number generators. The parameter ec represents the number of bytes of Reed Solomon error correction code that should be used on each droplet. The final parameters, stop and alpha, represent when the algorithm can stop generating droplets. Stop is a max number of droplets and alpha is a percentage of redundancy.

There are five methods associated with this class, not including the initialization method, and two of them are private helper methods. _calc_stop() is the first helper method and it examines the alpha and stop variables to determine the final number of droplets needed. If provided, stop will override alpha, otherwise, the number of redundant droplets needed is calculated with alpha. The second helper method is _update_seed() and it uses the lfsr object to get the next seed and set it for the random number generators. The three other public methods are make_droplet(), is_valid(droplet), and is_done(). Make_droplet() calls the _update_seed() method and uses the PRNG object to get a list of index values. The segments associated with each index value are retrieved and combined with an exclusive or (XOR) operator to generate the payload for the new droplet. The is_valid() method is then used to determine if a given droplet has successfully generated a valid nucleotide sequence. Finally, the is_done() method compares the number of valid droplets that were created with the final number of droplets needed as was previously specified. If the valid number is greater than the final number, then enough droplets have been created and the method returns true.

droplet.pyx

This file contains the class variables and methods for the droplet object, where droplets are created by the fountain algorithm. There are ten class variables associated with this object and five total methods, not including the initialization method, that are used by the encoding portion of the program. Currently, only the encoding program makes use of this class and the decoding program does not utilize this class. The first class variable represents the map object based of off the user provided mapping scheme. The next variable represents the seed value that is used to set the PRNGs for finding segment index values. Next is the payload variable. This is the

actual byte data that is to be stored in the droplet and represents multiple segments XOR-ed together. While the segments themselves are not stored in a droplet, a list of the segment index values is referenced by the segments_indexes variable. The degree, or number of segments contained in the droplet payload is also stored. The next two variables, ec and ec_obj represent the error correction variables. Here, ec is the number of bytes of error correction code to be appended to the seed and payload bytes and ec_obj is the Reed Solomon object that is used to find the actual bytes of the code. For the final class variables, the droplet converts its data, including the seed, payload, and error correction code into two different representations. The first, stored in the bytes variable is the integer representation of a list of bytes. The second, stored in the DNA variable is a sequences of nucleotide letters stored as a string. Then, if the mapping algorithm is able to convert the bytes to a valid nucleotide sequence, the valid variable is set to one.

The first two methods in this file are private methods and are used to create the different data representations of the droplet. The first, _to_DNA_strand(), converts the bytes to a nucleotide sequence based on the given map object. The second method, _package(), converts the seed, payload, and error correction code to a single concatenated list of bytes. The next three methods are getter methods for some of the droplet data. The first, get_byte_strand(), returns the bytes variable, while the second method, get_DNA_strand(), returns the DNA variable. The third getter method returns the valid variable that was previously set.

decode.pyx

This file includes the main method for the decoding part of the program. This file also has a method for parsing command line arguments and setting up usage feedback for the user. Instead of using a fountain object, the decoding program uses

a pool object. The main loop then reads in nucleotide sequences line-by-line from the given file. Each nucleotide sequence is processed by the pool object until enough are processed to decode the original file that was previously encoded. If successful, the decoded information is written to a file before the program ends. While the program is running, logging messages are printed to the output to update the user on the progress of the decoding process.

pool.pyx

This file contains the functionality for reversing the fountain codes algorithm to decode the original data. The pool class includes class variables and methods for decoding nucleotide sequences and uncovering the segments of the original data. Many of the parameters needed for creating a pool object are similar to the ones for creating a fountain object. The first parameter represents the number of segments that the final decoded data will contain. The second parameter, mapping, represents the map object that corresponds with the user specified mapping scheme. Seed_size is the size of the seed value, in bytes, so that a given sequence can split the seed from the rest of the data. This value is currently hard-coded to four bytes on the encoding side of the program. Data_size is the number of bytes represented in each segment. The delta and c_dist variables are the same as in the fountain object and are used to set up the random number generators. The ec variable is also the same as in the fountain object and represents how many bytes of the Reed Solomon code are contained in the sequences. Finally, the flag_correct variable is a boolean that allows the user to choose if they would like to analyze the error correction code or simply parse it out of the sequence.

The methods that make up this class include five private helper methods and five other methods for use by the decode.py main method. The first helper method,

called _check_ec(), checks the error correction code as is specified by the user defined variables. This method can check the Reed Solomon error correction code if it exists or simply remove and not analyze the code. If the code exists, is analyzed and too many errors are found, then the sequence is discarded and not used for the rest of the decoding process. The second helper method, called _parse_strand(), will split the the seed value from the data portion in a given sequence. The seed is then converted to its integer representation and returned along with the data section. The next helper method, called _add_droplet(), updates a dictionary data structure for use in the decoding process. This dictionary, called droplets_per_segment keeps track of all droplets that are associated with each segment index value. Since multiple droplets can contain a single segment, each segment index value can keep a list of droplets that contain the corresponding segment. This method attaches a single droplet to all index values contained in that droplets segments_indexes list in the dictionary.

The final two helper methods include a majority of the code for processing the decoding algorithm and removing segments from droplets and are called _update_entry() and _process_singles(). For a newly seen droplet, _update_entry() is called first. If the droplet contains one segment, the _process_singles() method is called right away. Otherwise, the droplet must have multiple segments so the method checks for any done segments that may be included in that droplet. If there are already done segments in the droplet, they are removed with the XOR operation and the droplet information is updated. If the droplet is left with one segment, it is added to the single segment queue called singles_q. Finally, a while loop in the _update_entry() method continues to call the _process_singles() method for each single segment droplet that is remaining in the queue. The purpose of the _process_singles() method is to set the single segments to the corresponding index in the done segments and then propagate that segment

throughout all other droplets that also contain the single segment. If a new single segment is recovered during the propagation process, it is added to the queue.

The next five methods in this file are called by code in the main method of the decoding file. The first of these methods, called add_dna(), includes code for processing a given nucleotide sequence. This method converts the given sequence into a byte array, checks for error correction, and splits out the seed and the data. The recovered seed is then used to set up a PRNG object in order to retrieve the corresponding number of segments and list of segment index values associated with that data. These values are added to a dictionary that keeps track of each droplet's information and the helper methods are called to process that information. Droplet objects are not generated here, as they are during encoding, in order to save on processing time and increase efficiency. If any errors occur during this method, a -1 is returned to the main, otherwise the seed value is returned. The next three methods are getter methods. The first, called num_seen_seeds() returns the number of seed values that have been processed while the second getter method, called num_done_segments() returns the number of segments that have been recovered out of the final dataset. The third getter method returns the list of all the recovered segments in the correct order and is called get_string(). This method was recently changed by discarding the map function call which was shown to take up too much processing time. Instead, the decoding file uses they bytearray() function. The last method of this file, called is_done(), returns a boolean value that is false if the number of done segments is less than the total number of required segments and true otherwise. This method signals to the main method whether or not enough processing has been done to recover the final data.

mapping.pyx

The main functionality of this code is to set up a mapping scheme given by the user. A user can provide their own mapping scheme, however, the code is designed to work with our specific mapping scheme described in Section 4.1. There are six class variables, three helper methods, and three public methods. The first two class variables are hard-coded lists of codons. One list represents valid codons that can appear in the mapping scheme and the second list represents illegal codons that should be excluded from the mapping scheme, even in the overlap of two concatenated codons. The third parameter, map_file represents the name of the user provided file with the mapping scheme. It is expected that the file converts hexadecimal characters to codons, where the hex is first, then a semicolon, then codons separated by commas. For example, each line would look something like this " 0 : AAC, GAC ". The last three variables represent dictionary representations of the map. A short map, which is the basic hex to codons, the full map, which includes previous codons, and a reverse map which maps codons back to their corresponding hex values.

The three helper methods of this class read in the map from the given file and set up the three mapping dictionaries. The three public methods then use those dictionaries to return the requested information to the user. These methods are get_codon_first(hex_val), get_codon(prev_codon, hex_val), and get_hex(codon). The first method would be used at the beginning of the translation process to retrieve the very first codon of the sequence. Therefore this method uses the short map and does not need a previous codon value. The next method for getting a codon uses the previous codon value to ensure that no illegal codon sequence is generated when the next codon is chosen. Finally, the last method is used for decoding codons back to their hexadecimal representation.

transpose.pyx

The code from this file is meant to be used in conjunction with map objects created from the mapping class. By utilizing the map object, this code represents the main functionality for converting bytes to sequences of nucleotides. There are two main methods, hex_to_codons(a, map_obj) and codons_to_hex(dna_str, map_obj) and one helper method, _append_codon(nt_sequence, codon, i). The first method, hex_to_codons, takes the given list of bytes and map object and returns a nucleotide sequence, or a -1. The -1 represents the inability of the algorithm to find a valid sequence within the set number of allowed backtracks. This method is responsible for generating sequences that do not include repeating sequences and have a GC content that remains between 45 and 55 percent. More information on this algorithm and the constraints can be found in Section 4.1. The helper method, _append_codon, can add the nucleotide letters from a codon into the main sequence and is utilized during the main translation process. The last method, codons_to_hex, is used during decoding and conducts the reverse functionality of hex_to_codons. Given the map object, a nucleotide sequence is converted back to a list of bytes.

lfsr.pyx

This file uses a Galois linear feedback shift register to generate seed values. These seed values are used by the Fountain Code to seed the PRNG object, in order to generate the needed data for droplets. This code was created by Yaniv Erlich for their Fountain Codes algorithm program and it was not changed for this implementation.

robust_soliton.pyx

This file contains the main functionality for the PRNG object. The code in this file creates a pseudo random number generator that can be seeded with a given value. Besides the helper methods for setting up the correct distribution, there are two

fundamental methods that are used by the fountain and pool code sections. This includes a method to set the seed value. It also has a method to return a number, d, from the Soliton distribution and a list of d numbers randomly generated from the total number of segments. This code was created by Yaniv Erlich for their fountain codes algorithm program and it was only slightly changed for this implementation.

APPENDIX B

REDNAM USAGE & EXAMPLE OUTPUT

Specific python libraries needed for this program include tqdm, reedsolo, numpy, scipy and Cython. The code can be compiled with the following command.

```
$ python setup.py build_ext --inplace
```

After cython code is compiled, the program can be run. For a help message, use the -h or –help flag:

```
$ python encode.pyx -h
```

which gives the following output.

```
usage: encode.pyx [-h] [--config_file] -f FILE_IN [-l SIZE]
                  [--delta DELTA] [--c_dist C_DIST] [--rs RS]
                  --map MAP [--stop STOP] [--alpha ALPHA]
                  --out OUT [--job_id JOB_ID]

Encode a given file to a list of DNA sequences.

optional arguments:
  -h, --help            show this help message and exit
```

```
  --config_file         parameters can be written to the
                        first line of the output file
  -f FILE_IN, --file_in FILE_IN
                        file to encode
  -l SIZE, --size SIZE  number of information bytes per sequence
  --delta DELTA         Degree distribution tuning parameter
  --c_dist C_DIST       Degree distribution tuning parameter
  --rs RS               Number of bytes for rs codes
  --map MAP             File that contains mapping scheme
  --stop STOP           Maximal number of oligos
  --alpha ALPHA         How many more fragments to generate
                        on top of first k (example: 0.1 will
                        generate 10 percent more fragments)
  --out OUT             File with DNA oligos
  --job_id JOB_ID       Used in conjunction with web app
```

An example run of the encoding program is shown below.

```
$ python encode.pyx -f one_mb.rand -l 32
     --delta 0.001 --c_dist 0.025 --rs 2
     --map original_map.txt --alpha 0.1
     --out one_mb.rand.dna
INFO:root:Reading the file. This may take a few minutes
DEBUG:root:Input file has 1048576 bytes
INFO:root:File MD5 is ed64ba6aa05e1adac70423061954a42c
INFO:root:There are 32768 input segments
```

```
INFO:root:Upper bounds on packets for decoding is 3
     4168 (x1.042732)  with 0.001000 probability

INFO:root:Finished. Generated 36045 packets out
     of 40172 tries (0.897)
```

A help message can also be displayed on the decoding program.

```
usage: decode.pyx [-h] [--config_file] -f FILE_IN
                  [-n NUM_SEGMENTS] [-l SIZE]
                  [--delta DELTA] [--c_dist C_DIST]
                  [--rs RS] [--map MAP]
                  [--no_correction] --out OUT

Decode a given list of DNA sequences to the original data.

optional arguments:
  -h, --help            show this help message and exit
  --config_file         parameters can be found on the
                        first line of the input file
  -f FILE_IN, --file_in FILE_IN
                        file to decode
  -n NUM_SEGMENTS, --num_segments NUM_SEGMENTS
                        the total number of segments in
                        the decoded file
  -l SIZE, --size SIZE  number of information bytes per sequence
```

```
  --delta DELTA         Degree distribution tuning parameter
  --c_dist C_DIST       Degree distribution tuning parameter
  --rs RS               number of bytes for rs codes
  --map MAP             File that contains mapping scheme
  --no_correction       Skip error correcting
  --out OUT             Output file
```

An example run of the decoding program is shown below.

```
$ python decode.pyx -f one_mb.rand.dna -n 32768 -l 32
     --delta 0.001 --c_dist 0.025 --rs 2
     --map original_map.txt --out one_mb.rand.dna.out
INFO:root:After reading 1000 lines, 1 segments are done.
So far: 0 rejections (0.000000) 1000 seen seeds
INFO:root:After reading 2000 lines, 2 segments are done.
So far: 0 rejections (0.000000) 2000 seen seeds
INFO:root:After reading 3000 lines, 8 segments are done.
So far: 0 rejections (0.000000) 3000 seen seeds
INFO:root:After reading 4000 lines, 16 segments are done.
So far: 0 rejections (0.000000) 4000 seen seeds
INFO:root:After reading 5000 lines, 20 segments are done.
So far: 0 rejections (0.000000) 5000 seen seeds
INFO:root:After reading 6000 lines, 22 segments are done.
So far: 0 rejections (0.000000) 6000 seen seeds
INFO:root:After reading 7000 lines, 24 segments are done.
So far: 0 rejections (0.000000) 7000 seen seeds
```

```
INFO:root:After reading 8000 lines, 29 segments are done.
So far: 0 rejections (0.000000) 8000 seen seeds
INFO:root:After reading 9000 lines, 34 segments are done.
So far: 0 rejections (0.000000) 9000 seen seeds
INFO:root:After reading 10000 lines, 37 segments are done.
So far: 0 rejections (0.000000) 10000 seen seeds
INFO:root:After reading 11000 lines, 43 segments are done.
So far: 0 rejections (0.000000) 11000 seen seeds
INFO:root:After reading 12000 lines, 45 segments are done.
So far: 0 rejections (0.000000) 12000 seen seeds
INFO:root:After reading 13000 lines, 45 segments are done.
So far: 0 rejections (0.000000) 13000 seen seeds
INFO:root:After reading 14000 lines, 53 segments are done.
So far: 0 rejections (0.000000) 14000 seen seeds
INFO:root:After reading 15000 lines, 63 segments are done.
So far: 0 rejections (0.000000) 15000 seen seeds
INFO:root:After reading 16000 lines, 78 segments are done.
So far: 0 rejections (0.000000) 16000 seen seeds
INFO:root:After reading 17000 lines, 85 segments are done.
So far: 0 rejections (0.000000) 17000 seen seeds
INFO:root:After reading 18000 lines, 105 segments are done.
So far: 0 rejections (0.000000) 18000 seen seeds
INFO:root:After reading 19000 lines, 130 segments are done.
So far: 0 rejections (0.000000) 19000 seen seeds
INFO:root:After reading 20000 lines, 147 segments are done.
```

```
So far: 0 rejections (0.000000) 20000 seen seeds
INFO:root:After reading 21000 lines, 164 segments are done.
So far: 0 rejections (0.000000) 21000 seen seeds
INFO:root:After reading 22000 lines, 169 segments are done.
So far: 0 rejections (0.000000) 22000 seen seeds
INFO:root:After reading 23000 lines, 180 segments are done.
So far: 0 rejections (0.000000) 23000 seen seeds
INFO:root:After reading 24000 lines, 201 segments are done.
So far: 0 rejections (0.000000) 24000 seen seeds
INFO:root:After reading 25000 lines, 242 segments are done.
So far: 0 rejections (0.000000) 25000 seen seeds
INFO:root:After reading 26000 lines, 322 segments are done.
So far: 0 rejections (0.000000) 26000 seen seeds
INFO:root:After reading 27000 lines, 487 segments are done.
So far: 0 rejections (0.000000) 27000 seen seeds
INFO:root:After reading 28000 lines, 563 segments are done.
So far: 0 rejections (0.000000) 28000 seen seeds
INFO:root:After reading 29000 lines, 672 segments are done.
So far: 0 rejections (0.000000) 29000 seen seeds
INFO:root:After reading 30000 lines, 756 segments are done.
So far: 0 rejections (0.000000) 30000 seen seeds
INFO:root:After reading 31000 lines, 988 segments are done.
So far: 0 rejections (0.000000) 31000 seen seeds
INFO:root:After reading 32000 lines, 1181 segments are done.
So far: 0 rejections (0.000000) 32000 seen seeds
```

```
INFO:root:After reading 33000 lines, 3711 segments are done.
So far: 0 rejections (0.000000) 33000 seen seeds
INFO:root:After reading 34000 lines, 9625 segments are done.
So far: 0 rejections (0.000000) 34000 seen seeds
INFO:root:After reading 34366 lines, 32768 segments are done.
So far: 0 rejections (0.000000) 34366 seen seeds
INFO:root:Done!
INFO:root:Writing to file
INFO:root:Done Writing file
```

Note that the input number of segments for the decoding parameter can be seen in the output of the encoding program. The user may need to know the amount of padding added to the file which can also be found in the output of the encoding program.