STRUCTURE AWARE SMART ENCODING AND DECODING OF INFORMATION IN DNA

by

Shoshanna Llewellyn

A thesis

submitted in partial fulfillment

of the requirements for the degree of

Master of Science in Computer Science

Boise State University

August 2022

BOISE STATE UNIVERSITY GRADUATE COLLEGE

DEFENSE COMMITTEE AND FINAL READING APPROVALS

of the thesis submitted by

Shoshanna Llewellyn

Thesis Title: Structure Aware Smart Encoding and Decoding of Information in DNA

Date of Final Oral Examination: 1 April 2022

The following individuals read and discussed the thesis submitted by student Shoshanna Llewellyn, 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
Edoardo Serra, PhD	Member, Supervisory Committee
William L. Hughes, PhD	Member, Supervisory Committee
Reza Zadegan, PhD	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 by the Graduate College by Tammi Vacha-Haase, Ph.D., Dean of the Graduate College.

ABSTRACT

Our increasingly information driven world is growing the demand for new storage technologies. Current estimates place the total storage demands exceeding the supply of usable silicon by 2040 [1]. DNA is an attractive technology due to its incredible density, almost negligible energy requirements, and data retention measured in centuries [1]. DNA does, however, come with new challenges. It is an organic compound with complex internal interactions which complicate the design and synthesis of DNA sequences for the purpose of data storage. In this work we demonstrate a new encoding-decoding process that accounts for some of the challenges in encoding and decoding, including issues arising from the secondary structure of the sequence, repeated nucleotides, unwanted subsequences, as well as GC content, vital for ensuring stable sequences. This is accomplished by using a graph representation of the possible encoding space that captures the relevant constraints, combined with a search algorithm that identifies the optimal encoding for the given input data accounting for these constraints. A benefit of our approach is that by leveraging the constraints on the encoding process, the decoding algorithm is able to correct single point errors without the aid of error correction codes; this is something no current competing solution can accomplish.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

The information age has brought an exponentially increasing demand for data storage. On our current course, demand for storage will eventually exceed our collective production capacity. In addition, storing all this information requires space and energy. Land to build storage facilities, while plentiful, is still finite, as is the energy needed to power them. Most of this energy is still currently produced from non-renewable sources. These factors are driving development into alternative storage technologies.

One promising solution is to use DNA as a storage medium. It is an incredibly stable compound. When properly stored it can theoretically maintain its integrity for centuries or millennia, far longer than conventional media. It also has much higher physical density. The entire human genome, roughly 3 billion base pairs of DNA, weighs roughly 3.6e-16 grams. This kind of data density is not currently possible with current or emerging technologies [1].

With these advantages, however, come numerous challenges. The primary issue this work attempts to address is that DNA is a biological compound that interacts with itself. These internal interactions complicate the synthesis process and can also introduce errors and instabilities. There are therefore two fronts on developing DNA data storage, the lab and technology side to improve synthesis and sequencing techniques, and on the software side to create new algorithms to account for DNA's properties as a material when encoding and decoding information.

It is not currently feasible to synthesize a string of DNA the length of an entire file, so every encoding scheme to date breaks the data up into "packets" and then encodes each

of these packets into strands of DNA. When decoding, the process is reversed, with each DNA sequence being decoded into its original packet, and then each packet recombined to form the original file. This work focuses on this packet encoding-decoding step: where a packet is translated into a single sequence of DNA.

These are the research questions asked by this work:

- How can we build a sequence that encodes specific information while also controlling for secondary structures?
- How can we find the shortest path between two nodes in a directed graph that has variable weights?
- Can constraints on DNA sequence design be used to correct insertion, deletion, and mutation errors?

These questions will be addressed and answered in Chapter 4 Methods and Chapter 5 Results of this work.

CHAPTER 2

BACKGROUND

2.1 Existing Data storage Technologies

The majority of current data storage is in magnetic tape, magnetic hard disks, and NAND flash. Magnetic tape and disk uses the magnetic alignment of ferrous particles to store information while NAND flash uses a voltage differential in a semiconductor. These are now very established technologies that are very well understood by the industry. However, there is a problem, as both of these technologies require energy to maintain and have a limited service life. Magnetic disks and tape wear out over time from mechanical stresses. Semiconductors slowly degrade as well, requiring higher and higher voltage differentials to maintain signal coherency until they simply do not work. These different storage mediums come with different costs in material and the energy needed to operate them. The resources needed to create storage to meet demands are finite. DNA data storage has promises to solve many of the problems that face existing technologies, most notably service life (how long the data can be stored for) and the energy required. A comparison of these different mediums can be seen in table 2.1.

2.2 The Challenges of storing data in DNA

The general mechanism for synthesizing a sequence of DNA is to start by making a single stranded sequence, and then use enzymes to create the second strand to complete the double stranded structure. One of the problems encountered with synthesizing DNA is

Table 2.1: Comparison of different memory technologies to DNA. Adapted from [1].

Memory ($type$)	Retention ($years$)	ON Power (W/GB)	Areal Density (bit/cm^2)	Volumetric Density (bit/cm^3)	Latency ($\mu s/bit$)	Error Rate
Flash	10 [1]	0.01 – 0.04 [1]	10^{10} [1]	10^{16} [1]	100 [1]	10^{-15} [2]
Hard Drive	> 10 [1]	0.04 [1]	10^{11} [1]	$10^{1}3$ [1]	$3 * 10^{3}$– $5 * 10^{3}$ [1]	10^{-15} [3]
Magnetic Tape	30 [3]	0.004 [4]	10^{9}– 10^{10} [5]	N/A	60 - 200 [5]	10^{-18} – 10^{-21} [3]
Cellular DNA	> 100 [1]	$< 10^{-10}$ [1]	10^{22}	10^{22} [6]	< 100 [1]	10^{-9} – 10^{-8} [7]

when it interacts with itself which can render the sequence unusable. Therefore, when designing DNA sequences to create some features must be avoided. These include (in no particular order) [8]

- Repeated nucleotides

- Repeated substrings

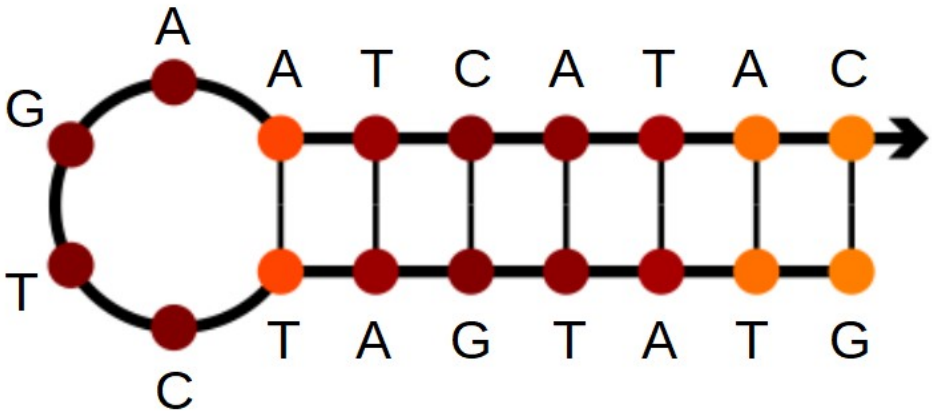
- reverse complimentary sequences (palindromes)

- GC content that deviates too much from 50%

Reverse complimentary palindromic sequences can create hairpins, where the DNA can bind to itself, as shown in figure 2.1. Repeated nucleotides of greater than length 3 can create instabilities in the sequence as well as make sequencing later more difficult. The same is true with repeated substrings. If the GC content deviates too far from 50% the sequence may break apart and become unusable [8] [1].

DNA sequences have a secondary structure defined by base pair binding, typically of complimentary bases between and within sequences (A-T, C-G). This can be predicted and modeled with energetics simulations based on the binding energies of these interactions [9]. The Nupack Project (www.nupack.org) is a software suite containing many tools for sequence analysis and design based on thermodynamic modeling [10]. One tool in this suite is called MFE, short for "Minimum Free Energy". This function predicts

Figure 2.1: Example structure of a hairpin caused by a palindrome



the most likely secondary structure among a set of possible structures based on what base pair bindings result in the lowest free energy of the system in kcal/mole [9]. As more bonds require more energy to break, a higher absolute MFE result means that the secondary structure contains more base pair bonds. If MFE = 0 kcal/mole, then the most stable secondary structure contains no base pair binding. This can occur if there are no secondary binding sites, or if the temperature is too high to allow bonds to form.

2.2.1 Summary of Previous Progress

So far, work in DNA data storage has successfully proven the concept, showing that it can be done. Examples are listed in the related works section. These works all involve actual laboratory experiments and prove the potential feasibility of this technology.

This thesis builds upon a previous work [11] where I contributed the idea and the relevant code to use codons as the encoding unit with a one-to-many encoding scheme, where one hexadecimal character could be coded with multiple codons. This is an imitation of the biological behavior of nucleic acids, where a single amino acid may be coded for by multiple codons. The prime contribution of this thesis is the use of a graph based encoding-decoding scheme and secondary structure prediction in the encoding

process.

2.3 Graph Theory

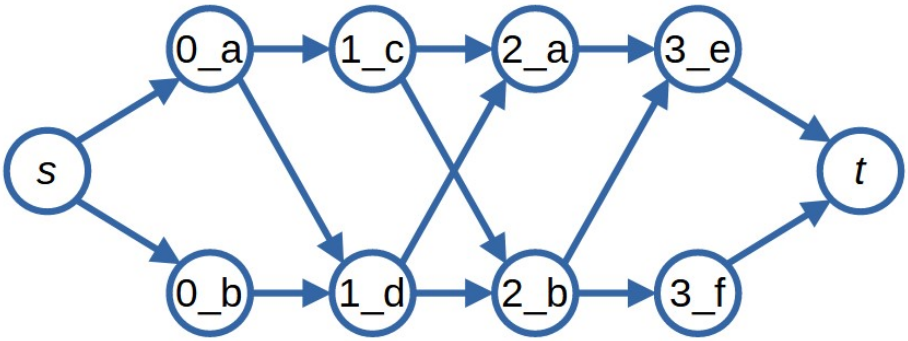
Graph Theory, as the name suggests, describes the study of graphs. Graphs are mathematical constructs that define the relation between different objects. These objects are referred to as "nodes" or "vertices" while the relation they have between other nodes are referred to as "edges".

There are several types of graphs. The ones relevant to this work are:

Weighted graphs - graphs where the edges have a weight describing the length, or cost, of that edge.

Directed graphs - the relation between nodes only goes in one direction. One way to think about this is a river flowing downhill. The water can only go one way. (in contrast, bidirectional graphs do not have this restriction)

Figure 2.2: Example of a unidirectional graph



Graphs are a common tool for modeling various problems in mathematics and computer science resulting in many tools to manipulate and traverse them. The most relevant to this work are algorithms for traversing graphs and finding the shortest path between two nodes within a graph.

2.4 Path Finding

There are many algorithms designed to find paths in a graph between two nodes. The type of graph influences which is most appropriate, as they all have their own advantages and disadvantages in terms of how they scale in memory and computational complexity based on the number of edges and nodes in the graph, as well as the presence of negative weights, loops, or if the graph is directional or not.

The two relevant algorithms are Dijkstra's algorithm and Uniform Cost Search. These are single source and single target algorithms. This means that the algorithm finds the shortest path from one vertex in the graph (the source) to another individual vertex (the target). There are other algorithms, such as Floyd-Warshall and Bellman-Ford, that instead find all shortest paths in a graph, or all shortest paths from one source vertex to all other vertices in the graph.

When discussing algorithms it is important to define their time and space complexity, or how long it will take to run and how much memory is required related to the size of the input. This is commonly notated with with $O(f(n))$, which is the upper bound or "worst case" performance. If an algorithm's worst case performance is also its best case performance (meaning it always runs in $f(n)$ time or takes $f(n)$ memory, then it is written as $\Theta(f(n))$.

Dijkstra's Algorithm [12]

Dijkstra's Algorithm (DA) is one of the most well known shortest-path algorithms. It works by recalculating the total distance starting from an initial approximation of infinity, finding the chain of nodes that results in the lowest distance.

```
Dijkstra(Graph, source)
    create priority queue Q
    n = number of vertices in Graph
    create array dist with length = n , all values = 0
```

```
create array prev with lengh = n, all values = undefined
for each vertex v in Graph:
    if v != source :
        dist[v] = inf
            Q.push (dist[v], v)

while Q is not empty:
    q = Q.pop # get the element with lowest priority
    if q = end:
        break
    for each neighbor p of q:
        d = dist[q] + length q to p
        if d < dist[v]:
            dist[p] = d
            prev[p] = q
            Q.update_priority(p,dist[p])

    return dist, prev
```

Dijkstra's worst case performance is $\Theta(|E| + |V| \log |V|)$, where $|E|$ is the number of edges in the graph and $|V|$ is the number of vertices.

Uniform Cost Search [13]

Uniform Cost Search (UCS) is a generalized form of Dijkstra's algorithm and is a form of "best-first search". It improves upon Dijktsra's algorithm where instead of all nodes being added to the priority queue, they are only added when their neighbor is visited. This makes it far less demanding on memory and scales better with larger graphs as the graph is only expanded as nodes are visited.

```
Uniform_Cost_search(Graph, source, end)
```

```
create priority queue Q
n = number of vertices in Graph
dist = 0
path = []
Q.push (dist, source, path )

while Q is not empty:
    dist,q,path = Q.pop # get the element with lowest priority
    if q = end:
        return dist,path
    for each neighbor p of q:
        new_dist = dist + length(p,q)
        new_path = path + p
        Q.push(new_dist,p,new_path)
```

UCS has the same computational cost as Dijkstra's, but because the queue is only expanded when nodes are visited, it has a lower memory footprint. UCS also has an added benefit in that it can be run "to completion", where instead of returning the first path discovered, it runs until the queue is empty and returns the best path. In this form the worst case computational cost becomes $O(b^{1+\lfloor C/\epsilon \rfloor})$, where b is the maximum branching factor of each node in the graph, C is the length of the shortest path to the goal, and ϵ is the minimum cost of each edge.

There are other algorithms that should be mentioned, but aren't applicable to the work of this thesis as they are not single source and single target algorithms.

Floyd-Warshall [14]

The Floyd-Warshall algorithm is an "all-pairs shortest path" algorithm, meaning that unlike UCS or Dijkstra's, it will find the shortest path between all pairs of vertices in a graph. It has a time complexity of $\Theta(|V|^3)$, where $|V|$ is the number of vertices in the

graph. This is both the best and worst case time complexity as it will always take this amount of time to process.

Bellman–Ford algorithm [15]

Bellman-Ford is a single source all shortest paths algorithm, finding the shortest path from one source to all vertices in a graph. It has a time complexity of $\Theta(|V||E|)$ where $|V|$ is the number of vertices in the graph and $|E|$ is the number of edges in the graph. As with Floyd-Warshall, this is the best and worst case time complexity.

Floyd-Warshall and Bellman-Ford aren't applicable to this work, but it was important to mention them as they are well known and fairly commonly used in graph related algorithms.

CHAPTER 3

RELATED WORKS

3.1 DNA Storage Algorithms

These works focus on how to store information into sequences of DNA. Information would be recovered by sequencing the DNA and translating those sequences back into the original information.

Early works in this field, such as that of Church [2] and Goldman [3] were spawned opportunistically by falling cost of generating synthetic DNA and sequencing. These works tend to be "proof of concept" showing the potential of storing information in DNA.

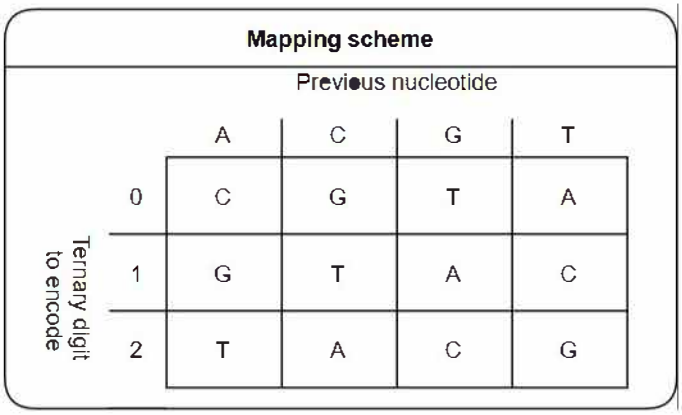
Church : Next-Generation Digital Information Storage in DNA [2]

This is one of the earliest works that combines next-generation synthesis and sequencing technology with an algorithm to store and retrieve a large amount of data in multiple DNA molecules. They used a very simple, yet functional algorithm to prove the feasibility of the concept of Nucleic Acid Memory, paving the way for future development. This algorithm is a "direct encoder", a term that will be used here to describe similar algorithms. Their direct encoder functions by mapping a 0 to "A" or "C", and 1 to "G" or "T". Bases were selected at random while preventing homopolymer runs of 3 or more nucleotides. Files were split up into separate sequences with unique index identifiers. In their experiments they experienced an unrecoverable bit error for every 0.7 megabytes of data stored. Part of this issue was that their error correction scheme could only account for mutation errors, not insertion or deletion.

Goldman - Towards Practical, High-Capacity Low-Maintenance Information Storage in Synthesized DNA [3]

Goldman's work was concurrent to Church's and uses a more complex translation scheme. One major departure is that instead of reading the input data as a binary sequence, it was read in base-3, or ternary code. This encoding scheme was direct, like Church's, but used a rotating code system (see figure 3.1) that used the previous nucleotide in the sequence to determine the choice for the next. This was done to prevent repeated nucleotides and reduce the chance for any patterns to develop. There is no attempt beyond this to control for patterns or any secondary structures, and there is no control on GC content in resulting sequences.

Figure 3.1: Rotating encoder used by Goldman *et al.*



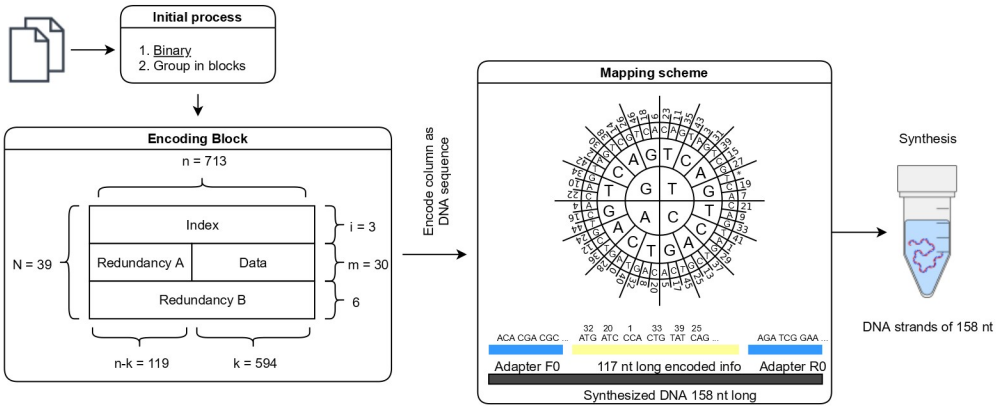
Another work, "A DNA-based Archival Storage System" [16] uses a similar rotating code mapping scheme. Their primary contribution was with a different error correction and file processing approach. They XOR consecutive data sequences together to create a third sequence. This allows the system to recreate one of the original two sequences if one is unrecoverable. The second innovation is to use unique DNA primers for different sequences. When DNA is synthesized, a primer sequence is appended. This sequence does not code for any data, but is used in the PCR (Polymerase Chain Reaction) process that is used to amplify the number of sequences to make sequencing possible. By using unique primers, they can selectively amplify sequences in the sample pool as a means

to perform random access of data. This highlights many of the developments of the 2010s, where most work was done on utilizing different physical technologies, with little exploration of computing and data theory.

Grass - Robust Chemical Preservation of Digital Information on DNA in Silica with Error-Correcting Codes [4]

Grass's work takes a biological inspiration from nature, using codons as an encoding unit. Their encoding process can be seen in figure 3.2. Files are first segmented into blocks. They are then converted to base 47 before being arranged into 713 by 39 matricies. These matricies consist of an index section, two redundancy sections, and the original data. When encoding, a column from each data block is read as a sequence and each base 47 character is then mapped to a unique codon. The adapter sequences on each side of this encoded sequence are constants and are there to aid in the sequencing process. There are additional interests of the encoding system used, but this particular one is of interest as it has the same original inspiration as the work of this thesis, nature and how DNA is used within a cell.

Figure 3.2: Encoding Process used by Grass *et al*.



Almost every following encoding scheme is either directly related to these early approaches, especially Goldman's (rotating codes) and Church (basic direct encoding). The main developments have been in the addition of newer error correction codes, algorithms

for splitting files into smaller sequences, and newer synthesis, sequencing, and storage technologies. In most of these works, the actual encoding algorithm tends to take a back seat, reflected in the relatively small focus it tends to get in the total work.

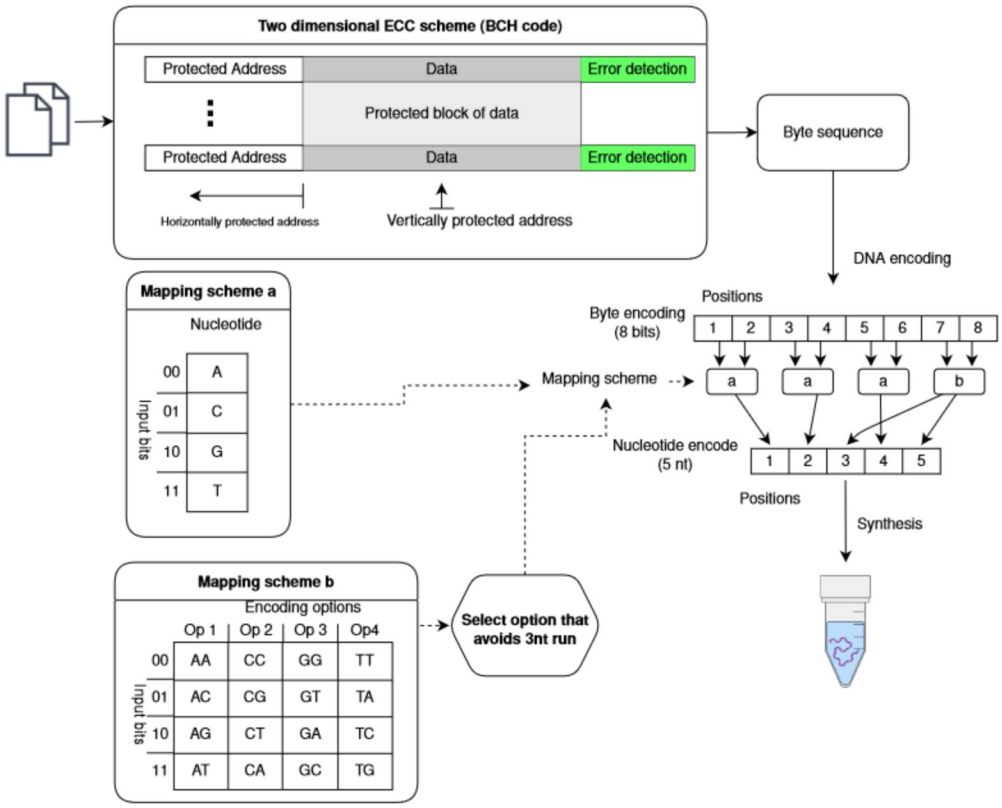
Organick - Random Access In Large-Scale DNA Data Storage [7]

This work's main goal was to develop ways to randomly access data in DNA sequences. This is because in other DNA storage algorithms the data must be completely decoded before being read. This feature is implemented in the data wrapping step, and not in the translation step. The key information relevant to the work of this thesis is the unique method they used for preventing repeated subsequences or patterns within the sequences. This was accomplished in two ways. The first was with a psuedo random number generator (PRNG) with a known seed. The PRNG is used to generate a bit sequence as long as the data sequence to encode and XORed with it to create a new data sequence. The second component is in a rotating code translation algorithm. This isn't described in the work, but it is assumed this is similar to the algorithm in Goldman's work [3]. "A Rewritable, Random-Access DNA-Based Storage System" [17] employs a similar system of XORing the sequence with the output of a seeded PRNG. Similar to the example with Bornholt and Goldman, most of their contribution to the field is in novel physical technologies, not in any new algorithms for encoding information.

Blawat - Forward Error Correction for DNA Data Storage [18]

This work is of particular note in relation to this thesis as it uses a choice based mapping scheme. The input sequence is read as bytes, and each byte is split into two pairs of bits. The first pair is directly encoded into a single nucleotide. The second pair is encoded as a pair of nucleotides, of which there are multiple options for every two bit pair. The option chosen is one that does not create a run of three nucleotides. This process is shown in figure 3.3.

Figure 3.3: Encoding Process by Blawat *et al*. The relevant portion are the two DNA mapping sections.



Erlich - DNA Fountain Enables a Robust and Efficient Storage Architecture [19]

Erlich's work isn't revolutionary in its encoder. It doesn't use a rotating code system like Goldman's work, nor a complex system like Grass. Instead, this work uses a simple encoder similar to Church. What they add is a constraint checking step where sequences are evaluated on GC content and repeated nucleotides. If they fail this constraint, the sequences are discarded. This process is possible because this work uses the Fountain Code algorithm to generate these candidate sequences from the source data.

Fountain Codes aren't a new idea in data transmission. Fountain Codes and derivatives are used commonly in wireless data transmission as they are resilient to data corruption. The idea is to first segment the data into sequences with unique identifiers. Two random number generators then work together to select sequences to be XORed together. The first generator determines how many sequences will be included in a

"droplet", the packaging unit of Fountain Codes. The second generator then determines which sequences will be included. Sequences are XORed together to create the droplet. A prefix is added to the beginning of the droplet which contains the number and identities of the sequences in that droplet. Finally an error correction code is appended to identify potential errors. In telecommunications, droplets are generated and transmitted until the sender receives a signal from the receiver saying that the message has been fully received. In this application that is obviously not possible, so a predetermined number of droplets are generated instead. The theory behind fountain codes is that if a message can be split into k sequences, then with fountain codes it should be possible to fully transmit the sequence in k droplets, even if some of the droplets are lost in transmission. Just in case, Erlich included a parameter to determine how many extra droplets to generate, as a percentage of the original number of sequences.

This approach comes with some advantages, but also a couple disadvantages. The main advantage is primarily in their software. Most of the algorithms mentioned here that have published their software have done so in a manner that is difficult to modify or tweak. The DNA Fountain Code work on the other hand is easily accessible, making it far easier to identify improvements and optimizations and reuse in other works. A second major advantage is that the fountain code system can be used to wrap around other encoding systems. The direct encoder they use does not have to be used, alternatives can be substituted in easily for straight forward comparison. And finally, the entire thing can be tuned and optimized further without fundamentally changing the algorithm. There are still disadvantages. The most obvious one is that because the entire premise is based on probabilities, one must check on their own if enough droplets were generated to decode the original data. The other major negative is that unlike other algorithms, which have a deterministic run time, this one does not. The culling system they employ means that it is theoretically possible for a given file and set of parameters for the program to simply loop forever and never finish. This scenario is, however, extremely improbable

and avoidable by not over restricting the encoder, but it is still indeed possible.

Suyehira - Using DNA For Data Storage: Encoding and Decoding Algorithm Development [11]

Suyehira's work was mentioned in the background section under section 2.2.1. The encoder combines a one-to-many encoding scheme based around codons with the Fountain Code packaging used by Erlich [19]. This thesis expands on this one-to-many encoder concept.

As the encoder processes the input sequence, it attempts to pick oligos to encode for a given hexadecimal without violating constraints (repeated nucleotides and GC content). If this is not possible, the encoder will backtrack and try again. To keep the algorithm from running for too long, there is a maximum number of backtracks allowed before the encoder will exit with a failure to encode. This failure is acceptable due to the nature of Fountain Codes. Much like with Erlich's work, if a droplet cannot be encoded, it will simply be discarded and a new droplet generated.

This work shares the same drawback of Erlich's fountain codes indeterminate run time. Other computational algorithms use backtracking, however, one of the goals of the work of this thesis is to create a new system that does not require such a crude mechanism.

One thing these algorithms all have in common is that they do not account for the secondary structure of the sequence when encoding data. They all depend on a combination of error correction codes and redundancy to account for this challenge. Most do account for the GC balance and repeated nucleotide problem in their encoding process, such as in the work by Church [2] and Erlich [19].

The increased interest in DNA data storage has caused new interest from outside the biotechnology and related fields, and now more computer science and mathematics related work is being published seeking to solve the more complex problems of DNA

sequence design. 2022 looks to be a year with even more works to be published on the subject.

3.2 Alternative Approaches to storing and retrieving information in DNA

Dickinson - An alternative approach to nucleic acid memory

Storing information in a single sequence of information is not the only approach being developed, and other methods are under investigation. These aren't related to the work presented in this thesis, but it would be incomplete to not mention that alternative ideas exist. The most important one of note is the idea of storing information in the complex structures formed by binding sequences of DNA. This idea is proposed in "An alternative approach to nucleic acid memory" [20]. The idea is to design sequences that form a self assembling two-dimensional structure. This structure can then be read with various microscopy technologies. This bypasses the issue with sequencing data, as microscopy can be done faster and the technology is already well developed with other uses pushing its improvement.

Löchel - Fractal construction of constrained code words for DNA storage systems

The work of Löchel et al. in "Fractal construction of constrained code words for DNA storage systems" [21] is interesting, as they demonstrate a novel way of analysing and creating new sequences of DNA for encoding information, what they refer to as "code words". Core to their idea is to represent a set of sequences as a fractal matrix, or "chaos game representation". This allows for a compact graphical representation of many complex patterns, or as they refer to them, motifs. These fractals are created by assigning each nucleotide a quadrant in a square and drawing vectors between them and new subquandrants based on a provided sequence. An example is shown in figure 3.4.

This fractal system is then used to create a frequency matrix which counts the frequency the vector path is in a quadrant, shown in figure 3.5. This means each element in the matrix would correspond to a different sequence and allows for a graphical representation of different patterns and sequence characterizations. For example, a representation of equal GC content would be having equal frequency in the top two quadrants and the bottom two quadrants.

Sheridan - Factorization and Pseudofactorization of Weighted Graphs [22]

This work explores using the factorization of a hybercube graph to express the encoding space of information into a sequence of DNA. This work is far more of a theoretical idea with no functional solution provided (such as a software artifact).

These two works are extremely new to the field and both show a new shift in DNA storage algorithms that attempt to solve the error prone nature of current technology of DNA synthesis, replication, and sequencing not with more error correction code, but in sequence design. This makes them interesting additions in the context of this work. Both of these works came out in December of 2021, and so were far too recent to influence the direction of this thesis.

Figure 3.4: Chaos Game Representation of the sequence C A T A G

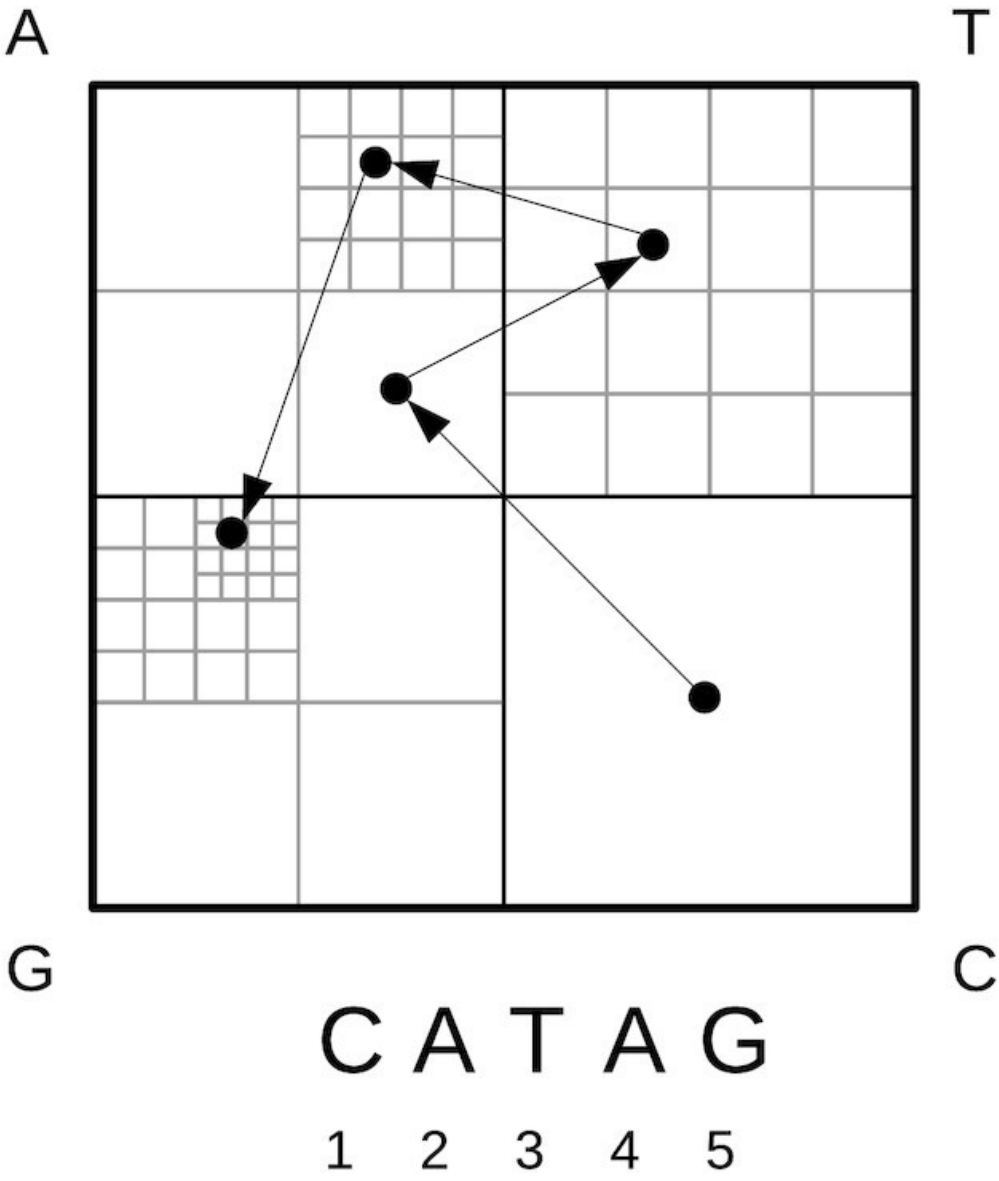
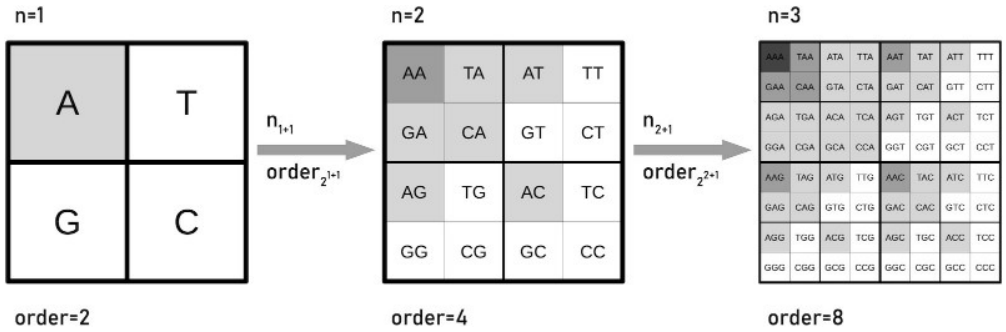


Figure 3.5: Frequency Chaos Game Representation Matrix of sequences of increasing scale



CHAPTER 4

METHODS

4.1 Overview

The general concept of this work is to create an algorithm where any provided input sequence of information has multiple viable DNA sequences which decode back to the same input sequence. This space of possibilities can then be modeled as a graph with a single source and sink such that any path from source to sink is a valid encoding. The different paths through the graph are determined by a mapping scheme which maps input data to nucleotide sequences, which in our case is a one to many mapping. Constraints are represented by static path weights in the case of local constraints, augmented by dynamic path weights to account for non-local constraints, such as repeat sequences and palindromes. The path searching algorithm then accounts for the local and non-local constraints of the DNA by choosing options in the graph path search that minimize the total path cost.

Decoding also uses a graph path search based algorithm. In the decoding case, the set of all possible DNA sequences that can be created is modeled as a graph. The path search algorithm then finds the shortest path through the graph to minimize edge weights and similarity score.

The specifics of how the encoding and decoding process works are detailed in their respective sections on "Static Elements" (section 4.3), "Encoding (section 4.4), and "Decoding" (section 4.5).

4.2 Glossary of Terms and Variables

- oligo = a short sequence of nucleotides, short for oligonucleotide. Typically used for whole sequences. In this section an oligo refers to a short string of nucleotides.
- n = number of nucleotides per encoding oligo
- h =number of bits per encoding oligo
- encoding unit = the unit in which data is processed
- l = number of encoding units
- G = graph
- d = input string
- p = number of oligos per character in the mapping scheme

4.3 Static Elements

Core to the encoding-decoding system is what will be referred to as the "Static Elements." These are static in that they are predefined and precalculated elements. For the encoder and decoder to function properly, they must use the same set of static elements. The static elements consist of the Dictionary and the Mapping scheme (or "Map"). These need to be created prior to any encoding or decoding.

First, in preparation for creating the dictionary, the number of nucleotides per oligo must be chosen. This value n will be 3 in all examples in this thesis. Once a value of n is chosen, the dictionary is created by generating all possible oligos of length n, and then culling those with three or more repeated nucleotides or other unwanted subsequences in them.

The dictionary itself can be represented as two matrices, B and W. B is a 1-dimensional matrix that contains a GC bias factor $B_i = (g_i + c_i) - n/2$ where g_i and c_i are the counts

of G's and C's within the ith oligo (oligo with a unique ID of i). W is a two-dimensional matrix, where $W_{i,j}$ is a weight factor that is determined by examining the concatenation of the jth oligo to the end of the ith oligo. Weight factors are defined by a lookup table of different oligos with an associated weight factor. For example, the table used in testing has an oligo "AAA" with an associated weight of 0.5 and another oligo "AAAA" with an associated weight of ∞. These weights were determined by reviewing relevant literature on synthesis and sequencing feasibility and given appropriate weights accordingly (Three repeated nucleotides can be tolerated more than four). $W_{i,j}$ is initialized with 1 and the concatenated oligo $i + j$ is compared to the lookup table. For every oligo in the table that is found within $i + j$ the associated weight is added to $W_{i,j}$. As an example, if $i =$"GAA" and $j =$"AAT", then $i + j =$"GAAAAT". $W_{i,j}$ starts as 1, add 0.5 as the $i + j$ contains "AAA" and then finally ∞ as it also contains "AAAA".

Once the dictionary is created, we determine the base,h, for how data will be interpreted by the encoder. This can be any positive integer value, such as 2 (interpreted as a stream of bits), base 10 (decimal) or base 16 (a string of hexadecimal characters). For all examples in this thesis $h = 16$. The next step is to create the map or mapping scheme. The mapping scheme, M, is a one-to-many scheme, meaning one bit-string of length h can be mapped to any positive number of oligos of length n. The number of oligos mapped per each encoding character in the mapping scheme is p. This map can be manually generated, or it can be defined by a generator algorithm. The generator takes the dictionary and creates a mapping scheme by first splitting the dictionary into two sets of oligos, one for oligos with high GC content ($B_i > 0$), and the other for oligos with low GC content ($B_i \leq 0$). This is done to ensure that there are options for each input character to balance the GC content when encoding. Then, for each character to be mapped, it randomly samples from each of these sets, without replacement, such that each character has an equal number of oligos with low and high GC content. Table 4.1 is an example of a possible auto generated mapping scheme using $h = 16, n = 3$ and is used in generating

the results of this thesis. In this mapping scheme each hexadecimal character is mapped to two oligos, one with high GC content, one with low GC content.

Table 4.1: Example Mapping Scheme

Hexadecimal	DNA
0	GCC, TCT
1	AGG, ACT
2	CAG, CAT
3	AGC, TAA
4	ACG, GAT
5	CGA, TTA
6	CTC, TAC
7	CGC, AGT
8	GTC, GAA
9	CCG, TTC
a	GCG, CTT
b	TCG, AGA
c	TGC, AAC
d	TCC, AAG
e	GCT, ACA
f	GCA, GTA

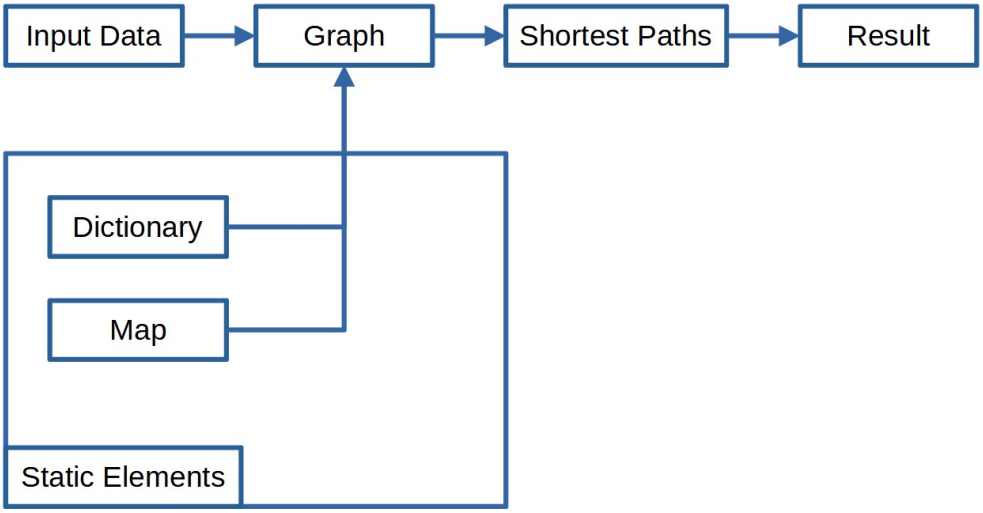
4.4 Encoding

Table 4.1 shows the process of encoding a byte sequence into a DNA sequence. The previous section described the static elements, and the next sections will describe how they are used to encode information into DNA.

4.4.1 Graph Creation

The encoding process takes the provided mapping scheme as discussed previously and a provided data string d with l characters of base h. This data string is then converted into a unidirectional graph representation of all possible ways that d can be encoded as a DNA sequence, d'. This graph, G, is initialized with non-coding source node s and non-coding sink node t. "Non-coding" means that these nodes do not translate to any oligo and are

Figure 4.1: Diagram of the encoding process



used only for the path search process. All other nodes in the graph are "coding" nodes as they represent an encoding of a single character from the input string to an oligo in the mapping scheme.

For each character d_i, $i \in \{0, .., l-1\}$, and a mapping $M(d_i, j), j \in \{0, ..., p-1\}$, a node $v(i, j)$ is added to the graph with a unique identifier (for the graph) of $i\_M(d_i, j)$. In this way, the graph can be described as having l levels of nodes where every node with an identifier starting with i are part of level i.

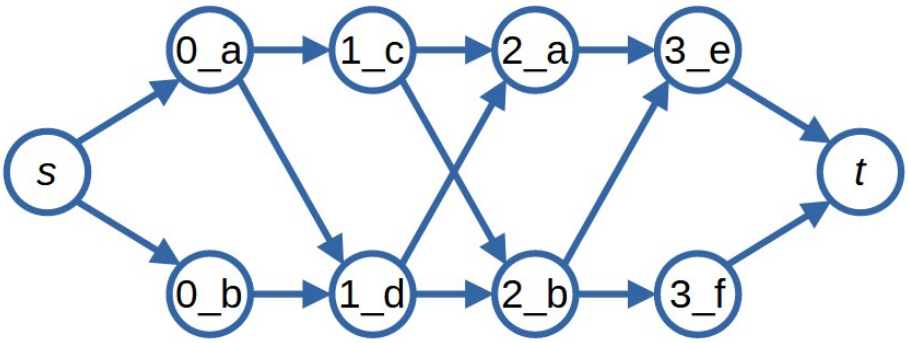
Edges are drawn from all nodes in level i to all nodes in level $i+1$. Weights are assigned to each edge based on the dictionary W matrix. If that weight is ∞ then the edge is removed. Lastly, the source and sink nodes are connected to the nodes in the first and last level of the graph respectively.

This results in a graph G_d. 4.2 is an example graph with 4 levels and a p value of 2.

This process has a computational cost and memory footprint of $O(l * h)$.

By comparison, a tree representation of the same space has a computational cost and memory footprint of $O(h^l)$, as it needs to hold every possible path in memory and compute it during creation. By contrast, this method shifts the majority of the computational cost to the path search algorithm and maintains a far lower memory footprint.

Figure 4.2: Example encoding graph



4.4.2 Graph Search

Finding the optimal encoding for a given input is accomplished by finding the shortest path through the graph using a modified Uniform Cost Search algorithm. This Uniform Cost Search is a modified form of the generalized UCS algorithm (as defined in the background material). There are two main modifications, one to the weighting algorithm and the other to the ending condition. In a conventional path search algorithm the cost of a path is equal to the sum of the costs on the edges in that path. In this algorithm, the cost function is $C = r + \sum w * g$ where C is the total cost of a path, r is a score based on the predicted secondary structures, w is the weight of the edge to connect to the next node in the path, and g is the GC ratio at each step of the path.

$P(v(i,j))$ is the path from s to node $v(i,j)$. The total cost of this path $C(P(v(i,j)) = \Sigma((w_x * g_x), x \in P(v(i,j)) + r(P(v(i,j))$.

This cost function was developed to balance the GC content, local constraints (repeated nucleotides), and the secondary structure. This last element and how it is calculated will be explored in 4.4.3.

This search algorithm operates until it has exhausted the priority queue, returning the best path it found. This means it has a total computational cost of $O(p^l)$. This was

Table 4.2: Example Queue 1

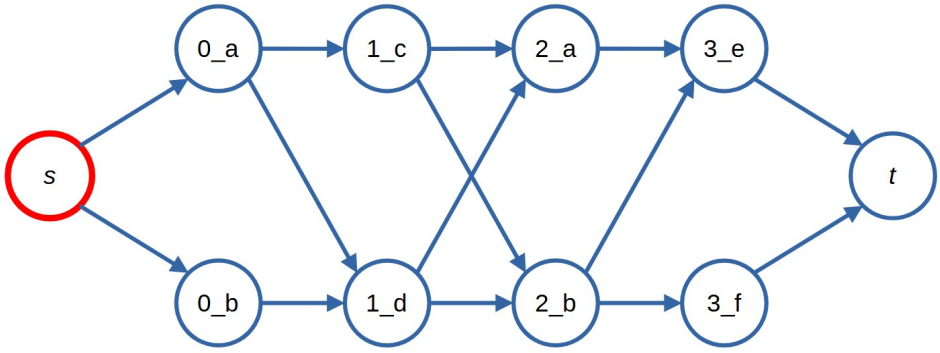
Cost	Node	Path
0	0_a	s, 0_a
0	0_b	s, 0_b

chosen to account for secondary structures, which can have non-localized impact on the viability of any given path. The problem can be described as saying the weights of the edges change based on the path taken.

The algorithm can be stopped early, when the first path is found, however this does not guarantee finding the best possible route. If stopped early, the computational cost becomes $O((l * p)log(l * p))$ (the same as Dijkstra's), and in this case it behaves like a typical UCS algorithm.

Using the graph shown in figure 4.2 as an example, here is how the process works:

Figure 4.3: Initial state



When the UCS is initialized, the source node 's' is put on the stack and the code proceeds to an iterative loop. This graph state is shown in figure 4.3

In the next step, 's' is popped off the queue as it is the only element. It then pushes two new entries to the priority queue. 0_a and 0_b. Since these are the first nodes and have the same edge weight from s, their priority will be the same. The queue at this state is shown in table 4.2

0_a was added first, so it is popped off the queue first, shown in figure 4.4.

Figure 4.4: Example Path: s $\rightarrow 0_a$

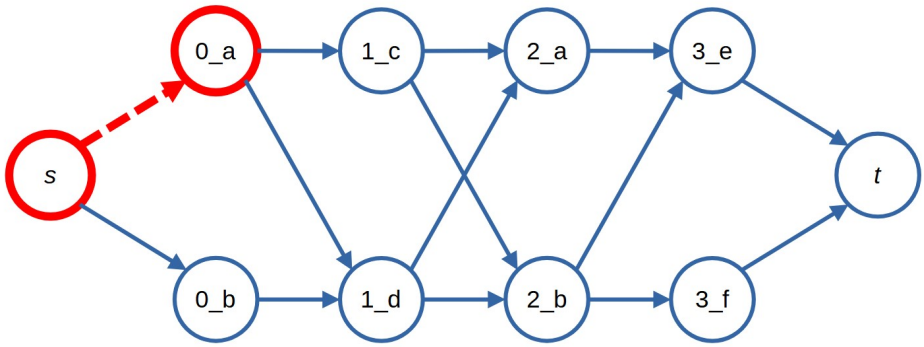


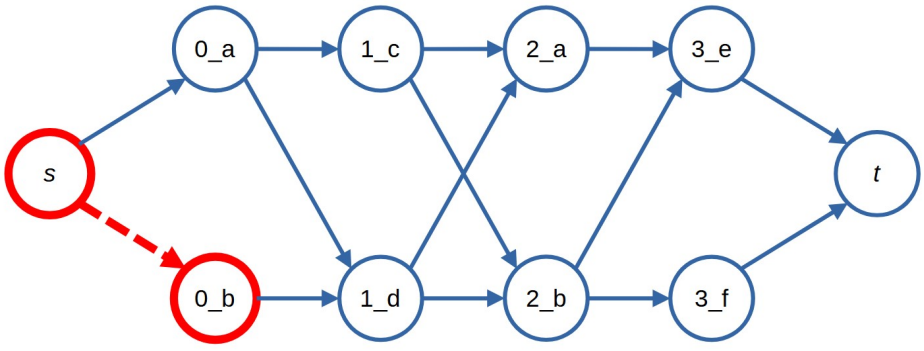
Table 4.3: Example Queue 2

Cost	Node	Path
0	0_b	s, 0_b
1	1_c	s, 0_a, 1_c
2	1_d	s, 0_a, 1_d

There are two neighbors to 0_a, 1_c and 1_d. They are added to the queue with a queue cost equal to a combination of the edge weights of the path, as well as a score based on predicted secondary structures. The queue now would look like table 4.3

The 0_b entry now has the lowest priority, so it is popped from the queue next. This is shown in figure 4.5 and the queue state in table 4.4

Figure 4.5: Example Path : s $\rightarrow 0_b$



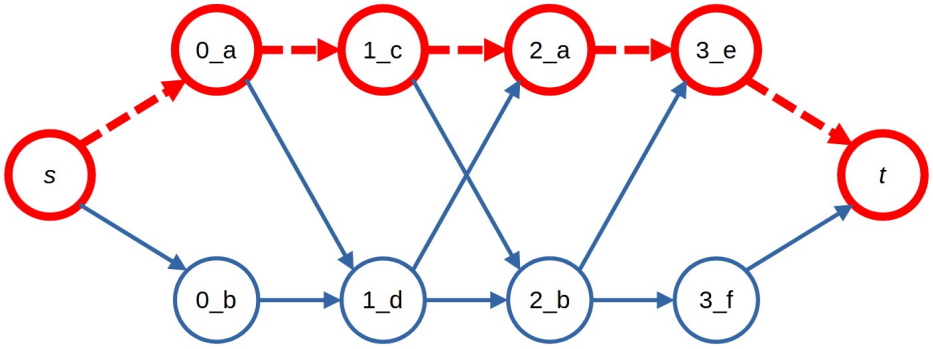
Like before, its neighbors are then be added to the queue, however there is only one neighbor, 1_d.

Table 4.4: Example Queue 3

Cost	Node	Path
1	1_c	s, 0_a, 1_c
2	1_d	s, 0_a, 1_d

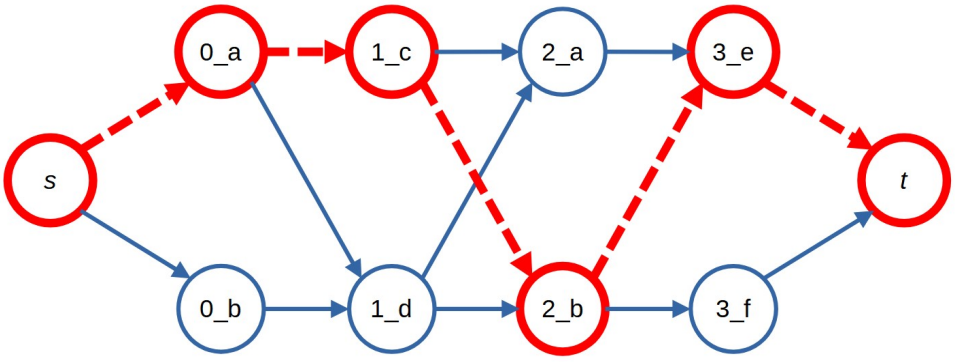
This process will continue until one of these paths reaches the sink node 't', shown in figure 4.6.

Figure 4.6: Candidate Path 1



In a traditional UCS algorithm, the entire process is complete at this point. However, in our algorithm, it is not. Because of the issue with secondary structures, we cannot be confident that the best path was found, so the best result is stored and the algorithm continues to explore additional paths, such as in figure 4.7.

Figure 4.7: Candidate Path 2



As an example, lets say that the path in 4.7 has an overall lower path cost than 4.6, but the path search finds the path in 4.6 first, because, for this example, it has a lower

static edge weight than the path in 4.7, even though it has a higher r score from its secondary structure. It is for this reason that the path search algorithm must continue until all possible paths have been found.

Eventually, all possible finite-cost paths to t are examined and the queue is empty. At this point, the algorithm returns the optimal path.

This modification, to examine all possible paths, is necessary due to the addition of the dynamic path score. In a typical UCS path search the path shown in 4.7 would have been completely missed and so a suboptimal path would have been returned instead. This modification and the addition of the dynamic score is necessary for accounting for the secondary structure of the resulting sequence d', as explored in the next section 4.4.3. This benefit does come with a trade-off on making the computational cost exponential. Due to this, the path search algorithm can be run in both modes: with and without dynamic scores and therefore in polynomial and non-polynomial time. The difference in quality of resulting sequences is explored in the results section.

4.4.3 Secondary Structure Prediction

The secondary structure of a DNA sequence, as mentioned in previous sections, can greatly impact the feasibility of synthesis. Therefore, being able to predict encoding paths that are more likely to create secondary structures and avoid them is extremely beneficial. Ideally a molecular dynamic simulation would be done, however, this is computationally extremely expensive and therefore not practical for this application. Thus an alternative is needed.

The problem with predicting secondary structures in an encoding algorithm is that, unlike with GC content or repeated nucleotides, or any unwanted subsequences, it cannot be simplified to a localized constraint. This is why a separate score is needed for cost calculation in the path search algorithm that takes into account the entire path, not just one branch in it.

The encoder uses the Nupack library for modeling the secondary structures of candidate sequences. Nupack is a library of functions for nucleic acid sequence analysis and design and uses binding energies to model how sequences of DNA or RNA interact with itself and other sequences. This work is not dependent on the Nupack library to function, and it can be replaced with other systems in the future.

Nupack contains a function called "mfe", standing for "mininum free energy", which returns the most likely secondary structure of the sequence along with the free energy of that structure based on binding energies of the different nucleotides. There are other functions that could be used, which return pairing probabilities, multiple potential structures, and more. This specific one was picked as it requires the least post-processing, returning a simple string that represents a secondary structure the provided DNA or RNA sequence could take, and still shows the benefit of this work as shown in the results when compared to when no secondary structure prediction is used in the encoding process. Specific settings used are in the supplementary material.

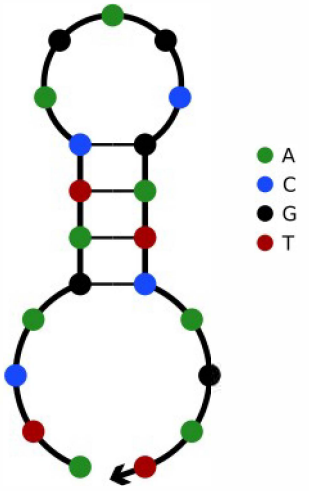
The output string of the "mfe" function is in 'dot-parentheses notation', where a parentheses is used to denote nucleotides that bind together, and a period for a nucleotide that are free and not bound to another nucleotide.

For example, take this sequence : "ATCAGATCAGAGCGATCAGAT".

For this sequence the mfe function returns the string "'....((((.....))))...." which represents this sequences predicted structure. This structure is shown in figure 4.8

This string is then fed into a scoring function. Each binding site is given a score equal to 10^k where k is the length of the binding site. The total score of the sequence is s which is equal to the sum of all binding site scores. The structure shown in figure 4.8 has a score $10^4 = 10000$, as it has one binding site involving 4 pairs of nucleotides.

Figure 4.8: Example result of "mfe" function



4.5 Decoding

Due to the nature of the encoding process, not all DNA sequences are possible. There are only so many oligos in the mapping scheme and there are constraints on how they can be arranged when encoding. This characteristic is exploited to add some error detection and correction for minor mutations.

Because the DNA sequence may or may not have errors, and the decoder has no knowledge of what the encoder originally produced, it first attempts to identify and correct for errors before the DNA can be decoded back into a bit string. This is done by identifying constraint violations in the sequence.

For decoding we will extend the glossary from the beginning :

- d' = the encoded DNA sequence (output from encoder)
- l' = length of the encoded DNA sequence
- e = the DNA sequence as provided to the decoder before any processing
- e' = the output of the decoder's graph based processing step, the "most likely" original sequence from the encoder

- $\overline{d}$ = the output of the decoder
- M' = all oligos present in the mapping scheme

If $|e| = l'$, meaning that the length is of the expected length, the algorithm will then continue on to the next step of the decoding process as normal.

This process follows the same graph path search idea used by the encoder. It starts by creating non-coding source and sink nodes s and t. From the mapping scheme M we get M'. For each level in the graph $i \in \{0, ...l-1\}$ and each oligo $M'(j), j \in \{0, .., |M'|-1\}$ a node $v(i,j)$ is added to the graph.

Next, e is split into l oligos of length n. Each node $v(i,j)$ in the graph is given a hamming distance score $u(i,j)$ equal to the hamming distance of e_i to the oligo corresponding to $v(i,j)$. The hamming distance is based on how many character mutations are required to turn the oligo $\overline{e_i}$ into the oligo of $v(i,j)$.

For example, if given a sequence of ATCGATCATCGT : the sequence is split into the following oligos : ATC, GAT, CAT, CGT. Within the mapping scheme is an oligo "GAT". At each level of the network, this oligo is given the following scores

0 (ATC) : 3

1 (GAT) : 0

2 (CAT) : 1

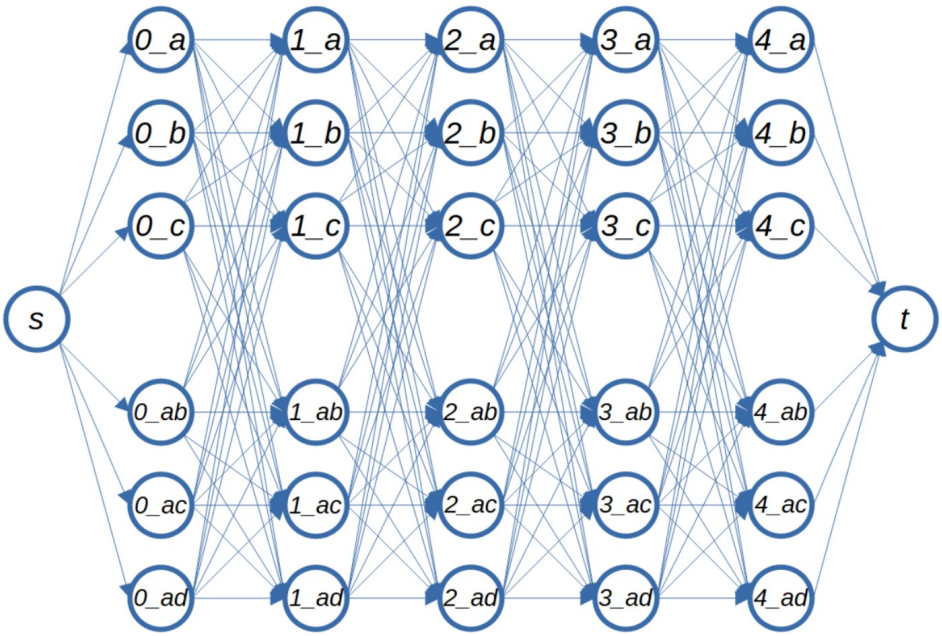
3 (CGT) : 2

Finally, edges are drawn from all nodes in level i to all nodes in level $i+1$ and assigned a weight based on the W matrix in the dictionary. Edges with ∞ weight are removed.

The computational cost and memory footprint of creating this graph is $O(h * l)$.

The Uniform Cost Search algorithm used remains largely unmodified from the original, only altering the standard weight function to include the edit distance scores of each node. The total cost of a path P is $C(P) = \Sigma(w_x) + q^{\Sigma(u_x)}, x \in P$, where q is a constant. This ensures that the chosen decoding is more affected by similarity to the input sequence

Figure 4.9: Example decoding graph



than by the weights, unless the weight is sufficiently high or the edit distance is extremely small. For this work a q value of 10^2 was chosen. Lower values ($< 10^2$) resulted in errors being introduced and higher values ($\geq 10^3$) provided no benefit, however increases the computational cost due to the nature of the programming language used (Python 3.x integers are unbounded).

The result of this is e', the most likely original DNA sequence created by the encoder. e' is then converted to $\overline{d}$, the most likely original data sequence that was encoded. This is done by taking each oligo in e' and finding its matching character in M. If there are no errors in this process then $\overline{d} = d$.

If $|e| \neq l'$, then there is at least one insertion or deletion error present in e. Correction is done by creating a set of new sequences by creating a set of new sequences E. If $|e| - l' > 0$ (the provided sequence is longer than expected) then $\overline{E}$ contains the set of sequences created by removing enough nucleotides to make the length of each sequence l'. The opposite happens if $|e| - l' < 0$. Each sequence in E is then processed by the graph based system and scored based on total path cost. The lowest scoring sequence is then

determined to be the most likely encoded sequence and is used as e' for the rest of the decoding process.

The current implementation of the decoder only attempts to decode if $||e| - l'| \leq 1$, as this gets extremely costly to compute the larger the difference. In addition, to reduce computational load, the implementation makes use of another optimization where it first attempts to identify a frame shift in e and only create sequences by altering the nucleotides in the surrounding region of the sequence. A frame shift is identified by finding the first oligo e_i that is not in the mapping scheme.

This process is not vital to the decoder or algorithm and was added to show the potential of this entire process, as no other algorithm known to date attempts to correct for insertion or deletion errors, and instead depends upon redundancy and other error correction codes. By reducing the dependency on error correction codes and redundancy, the chance for unrecoverable errors is decreased.

CHAPTER 5

RESULTS AND DISCUSSION

5.0.1 Introduction

All testing was done via simulation. Future work will be to validate these results with lab experiments using synthetic DNA.

Section 5.0.2 describes how the experimental simulations were setup

Section 5.0.3 describes the testing of the encoder

Section 5.0.4 describes the testing of the decoder

5.0.2 Experimental Setup

The dictionary and mapping scheme used for testing are mentioned in the previous sections, and copies can be found in the supplementary material. For all tests, $h = 16, n = 3$. These choices were made early on in the design process and is what most of the algorithm was built around. $h = 16$ means that the data can easily be processed as hexadecimal characters. $n = 3$ provides enough oligos in the dictionary to give every hexadeicmal character two oligos. Both of these values can be increased, but this has a negative impact on the computation time and memory usage.

5.0.3 Encoder Testing

The goal of this testing is to show how the encoder controls GC content and the secondary structure of the sequences. The main test is on randomized hexadecimal sequences. This is to simulate a realistic use case where data provided to the encoder is compressed.

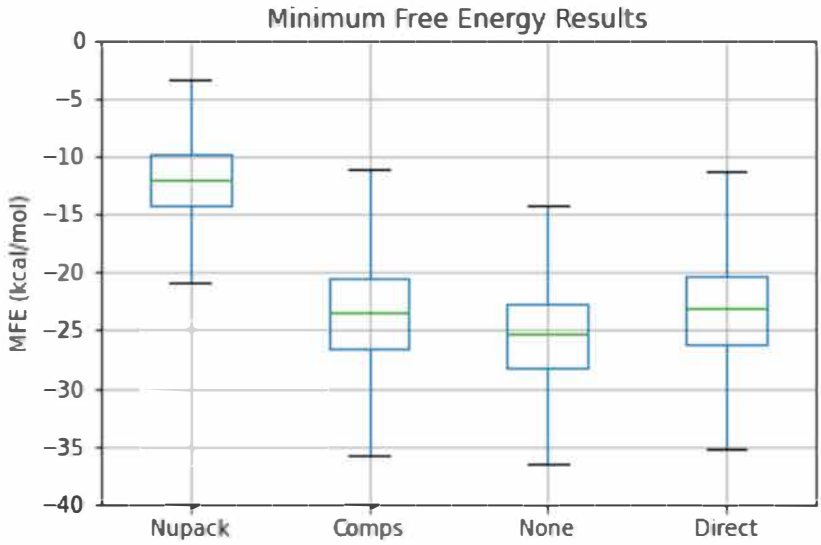
Four encoders are used in this test. The first three encoders are based on the work shown in the methods section of this thesis. One, labeled "Nupack" uses the secondary structure prediction and scoring system as outlined in section 4.4.3. "Comps" uses the same scoring system, but secondary structure is approximated with a reverse complimentary algorithm. This algorithm works by making a reverse compliment of the DNA sequence and finding the locations in both sequences that have the same nucleotide. "None" uses no secondary structure prediction, that component of the path cost is always 0. The secondary structure prediction algorithm change for "Comps" works by making a reverse compliment of the sequence, and then matching locations where nucleotides are the same in both the original and the reverse-compliment sequence. This is represented in the same dot-parentheses format the scoring algorithm uses to determine the *r* score of the sequence. Lastly, the "direct" encoder results uses a very basic direct encoder, the same used by DNA Fountain Codes [19].

To make the results of all encoders comparable, two datasets are used. Both are 100,000 randomized hexadecimal sequences. To make the output lengths match, "None", "Comps", and "Nupack" use sequences of 66 hexadecimal characters, while "Direct" uses sequences of 99 hexadecimal characters. This ensures that the output length of all encoders is 198 nucleotides long. The difference in data density is not relevant to this work, only the quality of the sequences.

Results of the encoders are then analyzed for "minimum free energy" with Nupack and for GC content. This is why output lengths have to be matched, as sequence length affects the MFE results, and shorter sequences will score better than longer ones. The results are presented as boxplots shown in figures 5.1 and 5.2 as box-plot graphs. The central line shows the mean, the the box encompassing the first through third quartiles of the data, and the whiskers showing the full range of the data.

An unexpected result is how "Comps" and "None" perform about the same as the direct encoder when it comes to secondary structure. It was unexpected as I thought

Figure 5.1: Secondary Structure Comparison - Minimum Free Energy

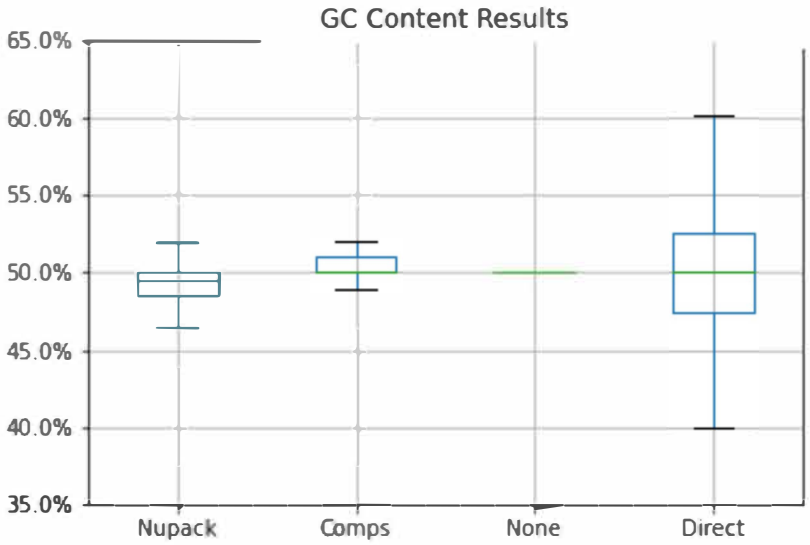


it would have some noticeable impact, even if it wasn't that much. This result shows that the additional processing with Nupack and path searching has a definite benefit on average sequence quality.

Recall from the background that deviation from 50% GC content increases the chance of errors in a sequence. In this test "None" performs the best, hitting exactly 50%. This is because in this example the GC content has a massive impact on path weight. "Comps" performs worse, showing that the alternative structure prediction system not only has no benefit to sequence quality, its worse than doing nothing. Lastly, the "Nupack" results in a slightly worse GC content on average, but still close to 50%. This is because of how G and C have different energies to A and T, so while the result is extremely good regarding MFE, it has a negative impact on the average GC content. Its still within acceptable tolerances, but this may indicate that additional tuning might be needed.

To see if this behavior is the same for more ordered data, Shakespeare's "Hamlet" was encoded as well. The text used was retrieved from Folger's Shakespeare Library [23] as a TXT file. The file is a 179kB ASCII text file. This experiment was done by converting into hexadecimal, and then splitting it into 5542 sequences 66 characters long, and for the

Figure 5.2: Secondary Structure Comparison - GC Content



direct encoder, 3696 sequences99 characters long. The results are shown in figures 5.3 and 5.4.

Figure 5.3: Secondary Structure Comparison - Hamlet - MFE

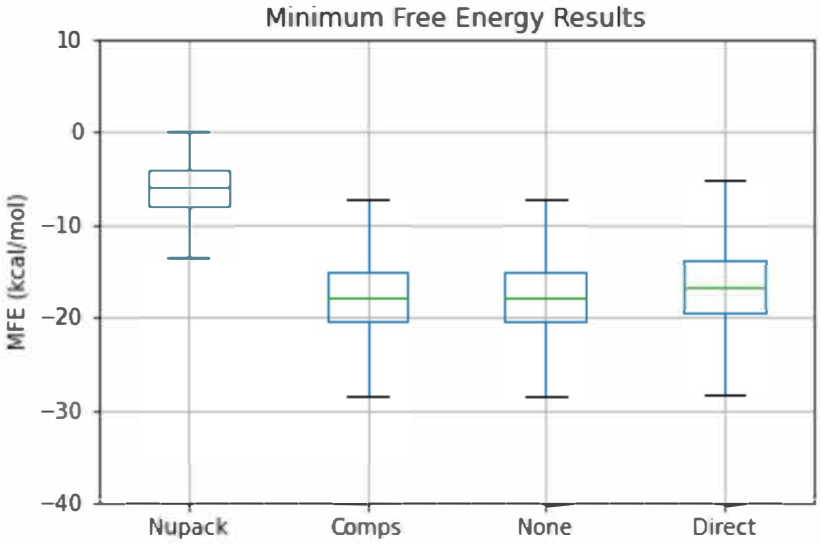
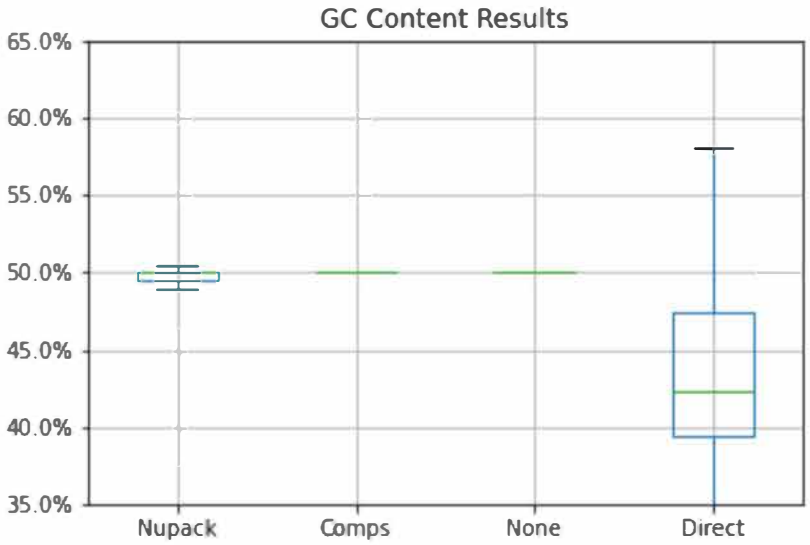


Figure 5.4: Secondary Structure Comparison - Hamlet - GC Content



These results follow the same trends as the above in regards to the MFE results. The GC content for the direct encoder is worse as the sequences this time aren't random and are far fewer. This result shows that even with uncompressed and more ordered data this encoder is still robust and able to create high quality sequences. The reason there is a difference from the results of the random data sets is down to the much smaller size and the more ordered nature of the data.

Fountain Code Testing

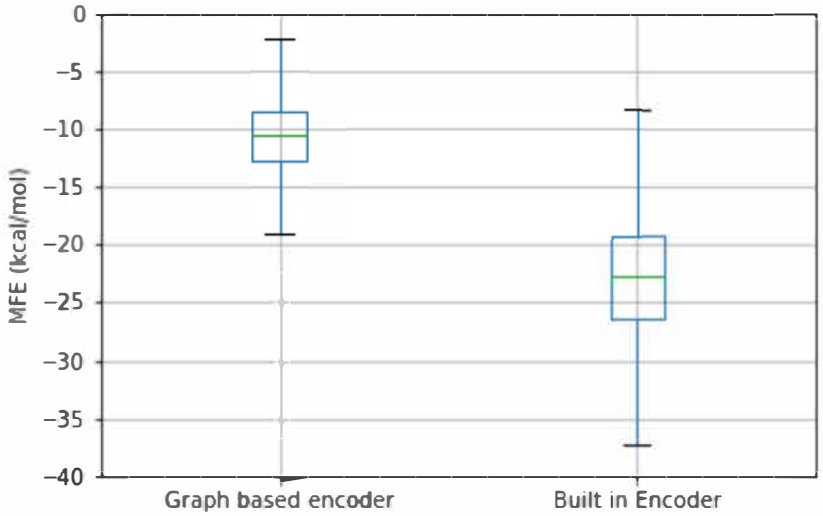
This test was done to test in a real application of encoding an entire file, and comparing this work to a pre-existing algorithm. For this, DNA Fountain Codes by Erlich [19] was chosen. Its a well known work in this field with a very open and easy to work with code base.

For this test "Built in Encoder" refers to the original functionality of the Fountain Codes algorithm with the only changes made to make it work in the testing environment.

"Graph Based Encoder" refers to a modified form of Fountain Codes, where the encoding process is cut short. In Erlich's DNA Fountain Codes algorithm, each droplet that is generated is checked for constraints and then, if passes, encoded in DNA. Otherwise, the droplet is discarded and a new one is generated. For this implementation, this entire step is skipped and every droplet generated is encoded using the "Nupack" based encoder from the above tests.

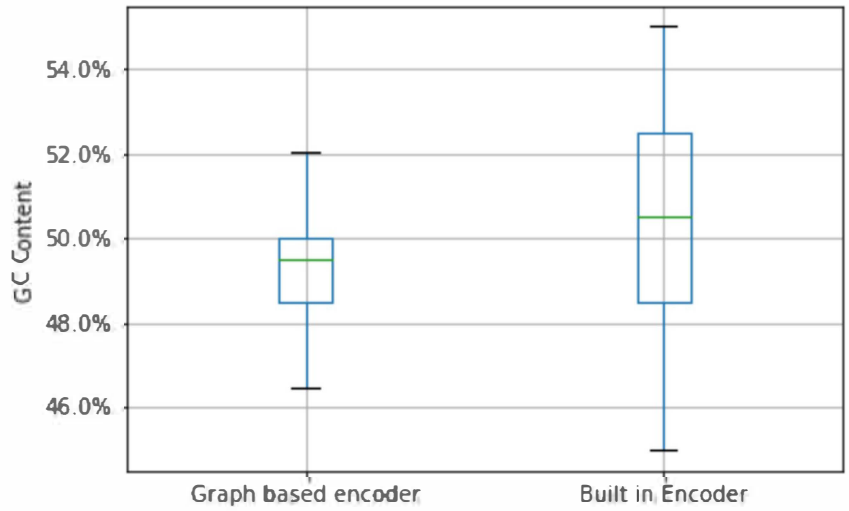
To make the results comparable, the arguments for bytes-per-sequence was adjusted to make the output sequence lengths the same. "Built in Encoder" was set to 46 and "Graph Based Encoder" was given 26. This results in output sequences of 196 and 198 nucleotides respectively. The results of this testing are shown in figures 5.5 and 5.6.

Figure 5.5: Encoding Results - Hamlet - Minimum Free Energy



Here we can see that the sequences generated by our encoder score better on MFE than the built in encoder. This would be expected as Erlich does not have any control for secondary structure, only GC and repeated nucleotides.

Figure 5.6: Encoding Results - Hamlet - GC Content



The result for the built in encoder is to be expected, as one of the parameters is the tolerance from 50% for GC content. Default is $\pm 5\%$ so these bounds are expected. Even still, the graph based encoder still produces a better result, with a tighter box plot.

5.0.4 Decoding and Degradation Testing

Decoding testing was done with the same encoder settings from the encoder testing using the nupack based secondary structure prediction from section 5.0.3. To first verify that the decoder functions as expected, all sequences were decoded normally with no errors introduced. This provided an expected result where all decoded sequences matched the original input sequences (meaning the decoded data matched the encoded data).

Next, for testing how the decoder responded to different types of errors, each dataset was duplicated three times: once for mutation errors, once for insertion errors, and another for deletion errors. In the mutation error datasets a nucleotide in a random location in every sequence was changed to another nucleotide at random. For the insertion error

sets this involved inserting a random nucleotide in a random location. And lastly, for the deletion error sets this involved removing a random nucleotide from each sequence.

This experiment were repeated three times for all 100,000 sequences in each data set with different seeds for the psuedo random number generator that was used to introduce the errors. The results from this experiment are shown in table 5.1.

Table 5.1: Error Decoding Recovery Rate

	Random Data Seet Average	StDev	Hamlet Text Average	StDev
Mutation	36.14%	0.017%	39.94%	0.607%
Insertion	37.53%	0.191%	35.48%	0.406%
Deletion	52.23%	0.103%	44.65%	0.557%

The first most obvious conclusion that can be drawn from this is that this encoding system is able to recover information even without error correction codes. It was able to accomplish this simply with knowledge of how the encoder is constrained. Moving on from the initial impressions of the results, there is some interesting behavior of note. The most obvious one is how different the results are between the random set and the hamlet set. This implies that the data structure itself has some impact itself on how well the decoder can recover, likely caused by the encoder behavior being differently constrained with un-ordered data (the random set) and ordered data (the hamlet set).

One consistency between both results is that deletion errors are far easier to recover from than mutations and insertion errors. The most likely reason is that deletion errors have a far greater impact on the sequence and are therefore far easier to identify, which would therefore make them easier to correct. Mutation errors have the possibility of either changing an oligo to another valid oligo, or shifting the sequence just enough that another sequence becomes a lower cost option in the decoder. Insertion errors add a frame shift, but also add potentially erroneous information that could confuse the decoder. On the other hand, deletions do not add any new information that can be misused, it only

destroys. This combined with a frame shift means that the decoder is more likely to identify the correct location of an error and the more likely correct nucleotide.

There are some problems with this test:

- It is not an exhaustive search over every possible error. That would simply be unfeasible due to immense computational requirements and unlikely to provide any benefit, considering the low deviations between different random seeds.

- These results cannot be applied to different mapping schemes or any other configuration changes. The only solid conclusion that can be extrapolated is that this algorithm is capable of decoding sequences with errors in them without any error correction codes.

- Only one error was tested. In mutation testing, only one mutation was added to a sequence, same for insertion and deletion testing.

CHAPTER 6

CONCLUSIONS

The goal of this work was to create an encoding scheme for storing digital information in DNA sequences with an encoding and decoding scheme that was modular and resilient to errors. These goals were both achieved. The encoding algorithm created is capable of optimizing GC content and secondary structures of the resulting sequences. In addition, the decoding side of the system is capable of recovering from some insertion and deletion errors without depending on redundant data, something that other comparable algorithms are not capable of. These outcomes show the power and potential of this graph based approach to encoding and decoding of information in DNA sequences.

Supplementary Material can be found here https://github.com/llewelsd/DNAMG

CHAPTER 7

FUTURE WORK

7.1 Fine Tuning

This algorithm has a large number of variables and elements that can be optimized. This includes weights for the dictionary, different mapping schemes, the scoring for secondary structures, and more. Optimizing these values was out of scope of this work. In future work on I would work to optimize the parameters which should provide better results than what was reported in the results.

7.2 Structure Prediction

The implementation of the algorithm explored here uses the Nupack library for predicting secondary structures. In future work this may be replaced with neural networks or some other predictive model that may be more computational efficient or more accurate and therefore provide better results. Preliminary work has already been completed on a transformer based neural network model for secondary structure prediction, however, this work is still in very early stages and better approach may present itself in future research.

7.3 Optimizations

There are many possible optimizations and application specific changes one can make to this algorithm that could not be explored within the scope of this work. In the future, I

would like to explore different weights, mapping schemes, and pathing algorithms to further improve upon the results of this work. In addition, optimizing the encoding and decoding process for different data packing schemes, such as Fountain Codes. One thought would be to alter the decoder to save broken (insertion or deletion errors) in a separate pool, and if needed attempt decoding.

7.4 Additional Improvements

Recent works on psuedofactorization of weighted graphs [22] and constraint visualization and construction with fractals [21] (as mentioned in 3.2) provide interesting options in improving weight generation, path searching, and decoding. The work of Lochel et. al. [21] is of particular interest in how it could be used to improve how edge weights are defined, or in path searching. Instead of depending entirely on just the predefined edge weights, this model could be used to define an additional heuristic score for avoiding repeated sub-sequences.

CHAPTER 8

REFERENCES

1. Zhirnov, V., Zadegan, R. M., Sandhu, G. S., Church, G. M. & Hughes, W. L. Nucleic acid memory. *Nature Materials* **15,** 366–370. https://doi.org/10.1038/nmat4594 (Mar. 2016).

2. Church, G. M., Gao, Y. & Kosuri, S. Next-Generation Digital Information Stor-age in DNA. *Science* **337,** 1628–1628. ISSN: 0036-8075. eprint: https://science.sciencemag.org/content/337/6102/1628.full.pdf. https://science.sciencemag.org/content/337/6102/1628 (2012).

3. Goldman, N. *et al.* Towards practical, high-capacity, low-maintenance information storage in synthesized DNA. *Nature.* ISSN: 00280836 (2013).

4. Grass, R. N., Heckel, R., Puddu, M., Paunescu, D. & Stark, W. J. Robust Chemical Preservation of Digital Information on DNA in Silica with Error-Correcting Codes. *Angewandte Chemie International Edition* **54,** 2552–2555. https://doi.org/10.1002/anie.201411378 (Feb. 2015).

5. Bornholt, J. *et al.* Toward a DNA-Based Archival Storage System. *IEEE Micro.* ISSN: 02721732 (2017).

6. Semiconductor Industry Association. International Technology Roadmap for Semiconductors, 2015 Results. *Itrpv* **0,** 1–37. ISSN: 0018-9162. papers2://publication/uuid/20F56C7C-3684-4039-B043-D3DE7C5293FA (2016).

7. Organick, L. *et al.* Random access in large-scale DNA data storage. *Nature Biotechnology* **36,** 242–248. https://doi.org/10.1038/nbt.4079 (Feb. 2018).

8. Ross, M. G. *et al.* Characterizing and measuring bias in sequence data. en. *Genome Biology* **14,** R51. ISSN: 1465-6906. http://genomebiology.biomedcentral.com/articles/10.1186/gb-2013-14-5-r51 (2021) (2013).

9. Dirks, R. M., Bois, J. S., Schaeffer, J. M., Winfree, E. & Pierce, N. A. Thermodynamic Analysis of Interacting Nucleic Acid Strands. *SIAM Review* **49,** 65–88. eprint: https://doi.org/10.1137/060651100. https://doi.org/10.1137/060651100 (2007).

10. Fornace, M. E., Porubsky, N. J. & Pierce, N. A. A Unified Dynamic Programming Framework for the Analysis of Interacting Nucleic Acid Strands: Enhanced Models, Scalability, and Speed. *ACS Synthetic Biology* **9.** PMID: 32910644, 2665–2678. eprint: https://doi.org/10.1021/acssynbio.9b00523. https://doi.org/10.1021/acssynbio.9b00523 (2020).

11. Suyehira, K. *Using DNA For Data Storage: Encoding and Decoding Algorithm Development* MA thesis (). https://doi.org/10.18122/td/1500/boisestate.

12. Dijkstra, E. W. A note on two problems in connexion with graphs. *Numerische mathematik* **1,** 269–271 (1959).

13. Wu, G. *Comparison between uniform-cost search and Dijkstra's algorithm* Aug. 2021. https://www.baeldung.com/cs/uniform-cost-search-vs-dijkstras.

14. Floyd, R. W. Algorithm 97: Shortest Path. *Commun. ACM* **5,** 345. ISSN: 0001-0782. https://doi.org/10.1145/367766.368168 (June 1962).

15. Datta, S. *Bellman Ford Shortest path algorithm* Oct. 2020. https://www.baeldung.com/cs/bellman-ford.

16. Bornholt, J. *et al. A DNA-Based Archival Storage System* in *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems - ASPLOS '16* (ACM Press, 2016). https://doi.org/10.1145/2872362.2872397.

17. Yazdi, S. M. H. T., Yuan, Y., Ma, J., Zhao, H. & Milenkovic, O. A Rewritable, Random-Access DNA-Based Storage System. *Scientific Reports* **5.** https://doi.org/10.1038/srep14138 (Sept. 2015).

18. Blawat, M. *et al.* Forward Error Correction for DNA Data Storage. *Procedia Computer Science* **80,** 1011–1022. https://doi.org/10.1016/j.procs.2016.05.398 (2016).

19. Erlich, Y. & Zielinski, D. DNA Fountain enables a robust and efficient storage architecture. *Science* **355,** 950–954. ISSN: 0036-8075. eprint: https://science.sciencemag.org/content/355/6328/950.full.pdf. https://science.sciencemag.org/content/355/6328/950 (2017).

20. Dickinson, G. D. *et al.* An alternative approach to nucleic acid memory. *Nature Communications* **12.** https://doi.org/10.1038/s41467-021-22277-y (Apr. 2021).

21. Löchel, H. F., Welzel, M., Hattab, G., Hauschild, A.-C. & Heider, D. Fractal construction of constrained code words for DNA storage systems. *Nucleic Acids Research.* gkab1209. ISSN: 0305-1048. eprint: https://academic.oup.com/nar/advance-article-pdf/doi/10.1093/nar/gkab1209/41767134/gkab1209.pdf. https://doi.org/10.1093/nar/gkab1209 (Dec. 2021).

22. Sheridan, K., Berleant, J., Bathe, M., Condon, A. & Williams, V. V. *Factorization and pseudofactorization of weighted graphs* 2021. https://arxiv.org/abs/2112.06990.

23. *Hamlet* July 2021. https://shakespeare.folger.edu/shakespeares-works/hamlet/.