Imperial College London

MEng Individual Project

Department of Computing

Imperial College of Science, Technology and Medicine

Decreasing Error Rates in DNA Data Storage through Constraint-Driven Convolutional Coding

Author:
Izer Onadim

Supervisor:
Thomas Heinis

Second Marker:
Holger Pirk

June 2023

Submitted in partial fulfilment of the requirements for Imperial College London's MEng in Computing.

Abstract

As rates of information production continue to increase exponentially, progressively denser data storage devices are needed. One promising approach is to store digital data on synthetic deoxyribonucleic acid (DNA). This option is appealing due to the incredibly high information density and durability of DNA. The process of storing digital data in nucleic acid requires a mapping between binary digits and the quaternary alphabet of DNA. This mapping, referred to as an encoding, must take into account various biochemical constraints, whilst trying to minimise errors and maximise information density.

In this work, an existing finite-state machine (FSM) based, constraint-driven encoding generation mechanism is evaluated and extended. The fundamental thesis presented herein is that this encoding scheme can be leveraged to correct certain categories of errors without any additional error-correcting codes (ECCs), whilst maintaining its original purpose of ensuring compliance with biochemical constraints. The novel results presented in this work demonstrate that, when faced with substitutions in particular, the encoding scheme is highly effective at detecting and correcting errors. Performance on other categories of errors is also assessed. This work additionally evaluates the different error-correcting capabilities of mappings produced by the scheme as they relate to factors such as symbol length, coding overhead and ability to meet biochemical constraints. Furthermore, this project proposes certain novel enhancements to the original encoding scheme and demonstrates that these changes allow the resultant encodings to account for previously unaddressed biochemical constraints. Other novel changes are proposed with the aim of decreasing the error rate, and the extent to which these amendments are successful is quantified. In support of these objectives, an efficient implementation of the amended version of the encoding scheme is also contributed.

Acknowledgments

I would like to thank my supervisor Dr. Thomas Heinis for his help and support throughout the production of this work. I would also like to thank Dr. Holger Pirk for his advice and feedback. I must express my gratitude to Dr. Omer Sella, whose Ph.D. dissertation provided the inspiration for this thesis, and who has been tremendously patient and helpful in answering my questions and offering suggestions to guide the direction of my work.

I want to thank my mother, Dr. Zerrin Onadim, for the many conversations in which she shared with me her expertise in the field of DNA and molecular biology, as well as all my family for their moral support throughout my degree.

Contents

Chapter 1

Introduction

Humanity's rate of data production is growing exponentially. Our ability to store data is increasing as well, but to a lesser degree. It is already impossible to store all the data we create, but this has not yet caused a problem since a certain amount of it is deleted, overwritten, or sensed without being stored for any significant period [1]. However, if current trends continue, the proportion of data we are able to retain will decrease.

When looking for new ways to store information, we may find inspiration in Richard Feynman's 1959 lecture, *There's Plenty of Room at the Bottom* [2]. In this famous talk, Feynman discusses the possibility of building "tiny machines" and creating "tiny writing". The examples he gives, such as attempting to write the entire *Encyclopædia Britannica* on the head of a pin, would most likely have seemed whimsical at the time. However, such information densities are no longer out of the question. Feynman then moves on to consider the issue of all the books in the world. He produces an estimate of 24 million volumes based on the contents of the American Library of Congress, the British Museum Library and the National Library in France, having accounted for duplicate copies. He calculates that if one were able to store a bit of information per 100 atoms, all these volumes would fit into a cube of material one two-hundredth of an inch wide. To demonstrate that this is no unattainable feat, Feynman points out that deoxyribonucleic acid (DNA) stores information at around one bit per 50 atoms. This leads to an interesting suggestion; what if we could use DNA to store digital data?

In 2012, Church *et al.* did just that[1]; they wrote 650KB of data to DNA and read it back with relatively few errors [3]. The maximum data density of DNA is around 10^{21} bits per centimetre cubed[2], several orders of magnitude greater than the densest widely-used storage media [4, 5]. Advancements in molecular biology have allowed us not only to sequence naturally occurring DNA but also to synthesise new DNA. For these reasons, DNA is an attractive candidate for digital data storage. There is a significant problem, however. Conventional storage media have read-/write latencies ranging from milliseconds to microseconds[3]; for DNA, such latencies are currently likely to be in the range of seconds to hours [6]. Synthesising and sequencing DNA involves many slow operations, which have only recently begun to be automated [7], so DNA latency is likely to never catch up to that of electronic media[4]. The relatively slow speeds of sequencing and synthesising DNA mean that it is likely to be utilised only for 'archival' storage purposes, in which data is written once and stored for many years or decades with a limited number of read accesses. This downside is balanced to a degree by the very high concurrency of all DNA operations; sequencing (reading), amplification (copying), and synthesis (writing) are all highly parallel, making for an estimated throughput on the order of kilobytes per second [6]. The costs of sequencing and synthesis have seen an exponential decrease [3]; if maintained, this decrease could make DNA an extremely attractive storage medium for archival workloads. Thus far, the primary applications for sequencing and synthesis technologies have all come from the life sciences, in which very high rates of accuracy are required; this has kept the cost of these procedures very high at several cents per nucleotide. In the field of data storage, the integration of error-correcting codes (ECCs) is commonplace, allowing for the accuracy requirements to be relaxed, and potentially decreasing the per-nucleotide cost to a sufficient degree to make DNA a viable archival storage medium.

In archival storage workloads, latency is not a priority; primary factors include density, durability, dollar cost per gigabyte read/written and energy cost at rest. Besides its legendary compactness, which allows it to store a theoretical maximum of 455 exabytes per gram [3], DNA has several features which make it suitable for archival storage. DNA is highly durable - when stored in bone with no treatment to improve durability, it has an estimated half-life of approximately 520 years [8]. It has been seen to last considerably longer in cold, dark, and dry environments, as evidenced by the 2013 recovery of a horse genome from a bone trapped in permafrost for 700,000 years [9]. Current storage media, such as hard disk drives and magnetic tape, are rarely given warranties of more than five or ten years. This led one of the pioneers of DNA storage, Nick Goldman, to remark,

[1] Though they were not the first to do so. A brief history of DNA storage can be found in section 2.1.

[2] The issue of finding a precise information density for DNA is difficult; one must make assumptions about the form in which the DNA is stored, whether it is double or single-stranded, and how tightly it is packed. The figure quoted is the maximum density that can be achieved when storing double-stranded DNA in a dried, crystal form, according to Zhirnov *et al.* in [4]. However, as pointed out by Zhirnov *et al.*, the operations of reading, writing and copying DNA require other components, so it may be more realistic to assume an information density similar to that of a cell such as *E.coli* ($\sim 10^{19}$ bits/cm^3 [4]), which is still considerable. If stored in single-stranded form, higher densities may be possible.

[3] Temporary memory units such as RAM and registers have nanosecond access times, but these do not persist data.

[4] Based on the cell division of *E.coli*, we can estimate a best-case read/write latency on the order of 100 μs per bit [4], but it is likely that electronic media will have achieved much lower latencies by the time automated sequencing/synthesis technologies approach this limit.

> *"In data centres, no one trusts a hard disk after three years. No one trusts a tape after at most ten years. Where you want a copy safe for more than that, once we can get those written on DNA, you can stick it in a cave and forget about it until you want to read it."* [10]

Moreover, DNA has a meagre cost when at rest, i.e. when not being accessed, since the conditions needed to store it safely are easily and cheaply achieved[5]. This low storage cost goes part of the way towards balancing the very high cost of DNA synthesis and sequencing, but a continuation in the trend of decreasing costs is still required to make DNA storage a viable option for data centres and archives.

DNA has another, very unique advantage; it is unlikely to suffer from obsolescence like other storage media. Floppy disks and VHS tapes, for example, are currently rarely used. The technology for reading these devices is no longer being improved and may one day be forgotten (or at least become very expensive and difficult to attain). This obsolescence factor means that archives must constantly be maintained and updated, copied and transferred from old media to new, at a high cost to their maintainers. DNA, the molecule encoding all genetic information, which has seen ever-increasing utilisation in medicine and biology, is unlikely to suffer the same fate. It seems reasonable to assume that for as long as there are biological humans with DNA, the ability to sequence and synthesise it is likely to be maintained and improved, especially with the advent of genetic engineering.

In order to store digital data as DNA, a function must be defined to map a string of binary data to a sequence of quaternary nucleotides. This mapping is known as an encoding, and it will often integrate indexing information, error-correcting codes, and possibly also primer sequences for random access into the resultant nucleotide sequence. This sequence is then synthesised[6] into a strand of DNA which can then be stored, usually *in vitro*. After storage for an arbitrary length of time, the data encoded in the DNA can be accessed via DNA sequencing, for which there are a number of techniques. At this point, the quaternary string of nucleotides can be decoded back into binary data, and error-correcting codes can be applied to detect and correct any errors that have arisen. We can refer to this sequence of steps as the DNA data storage "lifecycle", which will be examined in greater depth in chapter 2.

When designing an encoding to map binary to quaternary data, there are many constraints that must be considered - it is usually insufficient to simply map each of the four two-bit sequences[7] to one of the four nucleotides. Such a naive encoding will result in a nucleotide sequence that is error-prone. This is because the various techniques for performing synthesis, amplification and sequencing experience higher error rates when writing/reading certain base sequences. An example of such a sequence is a homopolymer - a subsequence consisting of more than one consecutive repetition of a single base, e.g. AAAA or CCC. Homopolymers are prone to PCR[8] slippage, which can result in deletion errors, so it may be wise to use an encoding that avoids homopolymers longer than a certain maximum length. There are many other similar constraints which an encoding may take into account.

An important consideration is that, as pointed out by Sella *et al.* in [13], constraints are likely to change as techniques for reading, writing and copying DNA develop. Similarly, differing aims may lead to differing requirements; for example, when synthesising DNA for random access, a primer sequence is added to identify and extract different sections of stored DNA; this primer sequence would need to be reserved such that the encoding avoids producing it in the data. It is not difficult to imagine that different DNA storage applications in the future may need further reserved sequences. The result of this is that as constraints and applications change, the encodings used will become obsolete and will need to be updated. In response to this problem, Sella *et al.* propose an encoding generation mechanism, which can be used to create an encoding based on a given set of constraints. This will allow for different encodings to be generated using different constraint lists, catering for the requirements of various DNA data storage systems and simplifying the process of adapting to changes in the underlying technologies. A more detailed overview of the encoding generation mechanism is provided in chapter 3. This thesis builds upon the work of Sella *et al.* by first providing further analysis of the encoding generation scheme and then using this analysis to inform extensions and amendments to the encoding generator.

[5] As pointed out by Goldman *et al.* in [11], the Global Crop Diversity Trust's Svalbard Global Seed Vault would be a perfect environment to store DNA, despite having no permanent on-site staff, due to it being cold, dark and dry [12].

[6] DNA synthesis is usually performed by one of two processes, both of which are explained in greater detail in the background section.

[7] The four permutations of two binary bits: 00, 01, 10, 11.

[8] Polymerase Chain Reaction. This is a fast and efficient process for amplification, i.e. making a very large number of copies of a DNA strand.

1.1 Thesis and Contributions

The principle aim of this work is to answer the question "Can a finite-state machine (FSM) based, constraint-driven encoding scheme as presented by Sella *et al.* be leveraged to mitigate errors which arise in the archival DNA data storage life-cycle?".

The analysis performed herein clearly demonstrates that certain categories of errors can be successfully detected and corrected using such an encoding scheme, and examines the different error-correcting properties of various parameters that may be used by the scheme in the generation of a particular encoding. In support of this analysis, we also contribute an efficient implementation of the encoding scheme as it is described in [13]. Having assessed the error-correcting abilities of the standard encoding scheme, we go on to extend it in the hope of improving its error-correcting performance and its ability to comply with biochemical constraints. We then analyse the enhanced scheme in a similar fashion to the original one and demonstrate that, whilst the error-correction performance is unchanged, the enhanced scheme can meet two previously unmet biochemical constraints without increasing coding overhead. Therefore, the contributions of this work can be summarised as follows:

1. An analysis of Sella *et al.*'s encoding scheme, which demonstrates its effectiveness as an error-correcting code. This analysis explores how the parameterisation of the encoding generator affects the efficacy of the produced encodings, revealing which configurations lead to better error correction performance at different levels of coding overhead. This analysis can be found in chapter 5.

2. An extension of the encoding mechanism, including further analysis of the new version of the scheme. This extension presents two novel techniques for meeting biochemical constraints which the original scheme did not address. These extensions are presented in chapter 6.

3. A new, efficient implementation of the extended encoding generation scheme, as well as a framework for generating and experimenting with different encodings[9]. The implementation of the original encoding is presented in chapter 4. The framework used to experiment with different encoding configurations is covered in chapter 5. The implementation details of the various extensions can be found in chapter 6.

Novel Elements

The most interesting novel elements in this work are: (1) the discovery that the encoding generation scheme of Sella *et al.* can be used (with minor modifications[10]) to generate error-correcting encodings; (2) a set of results demonstrating which encoding configurations produce better error-correcting performance and some subsequent recommendations on how to manage the trade-offs between speed, error-correction and coding overhead; (3) novel enhancements to the encoding generation scheme, which enable it to meet two previously unmet biochemical constraints[11] without increasing coding overhead and whilst maintaining error correction capacity.

The novel results found in this work have also been prepared as an article for publication, which can be found in appendix D.

[9] The implementation is made freely available under an MIT Licence and can be accessed at `https://github.com/IzerOnadim/dna-storage`.

[10] The minor modifications made to the original encoding (not including the extensions discussed in chapter 6) are that, in this work, the number of reserved bits is set manually as a hyperparameter to control the coding overhead, and that the random choice mechanism is used (whereas Sella *et al.* did not prescribe the use of any particular choice mechanism in [13]).

[11] These two constraints are limitations on GC variation and short-tandem repeats, both of which are partially prevented by the new suggestions.

Chapter 2

Background

Deoxyribonucleic acid (DNA) is a double-stranded molecule that acts as the medium of storage for the genetic information of all living creatures[1]. A strand of DNA is formed from a sequence of building blocks called nucleotides, each consisting of a phosphate molecule, a sugar molecule, and a nitrogenous base, i.e. a basic, nitrogen-containing molecule. Each nucleotide connects to its neighbouring nucleotide through the interaction of its sugar molecule and its neighbour's phosphate molecule; hence a strand of DNA is held together by a 'backbone' of phosphate-sugar bonds to which bases are attached. Nucleotides are identical to one another except in their nitrogenous base, which can be one of four compounds: adenine, guanine, cytosine and thymine, abbreviated as A, G, C and T[2]. It is these bases that act as the information carriers; the sequence in which different bases appear determines the data encoded by a DNA strand. Pairs of bases can become linked through hydrogen bonds, allowing two strands to bind together in a double helix to form the double-stranded structure of DNA. Each base will only bond with its complement; adenine with thymine and guanine with cytosine. Therefore, the base sequence of one strand of DNA can be wholly determined by the strand to which it is bonded [14, p. 4]. An illustration of this structure can be seen in Figure 2.1.

The different nucleotides found in a DNA strand form a quaternary code - a language with an alphabet of four letters - much in the same way as computers make use of a binary code. In the 'coding' sections of DNA, known as exons[3], the nucleotides are grouped into threes. Each triplet, referred to as a codon, represents a different amino acid, which are the building blocks of proteins. This mapping of a triplet of bases to an amino acid can be thought of as the DNA encoding used by nature. This encoding is highly suitable for the biology of living creatures since all our proteins consist of 20 naturally occurring amino acids [14, p. 7]; thus, three letters of a quaternary code are the minimum needed to represent all the different amino acids[4]. Since our aim is to store digital data in DNA, we are not restricted by nature's code; we can pick any arbitrary encoding. For example, we could map each nucleotide to two bits. Or we could organise our nucleotides into sequences of, say, five bases, which together form a symbol (similar to a codon, but longer), and then have each symbol code for a byte of digital data. Each of these encodings may have different properties in terms of information density, GC content, and the presence of short tandem repeats and homopolymers, so the decision on which encoding to use should be informed by the constraints and requirements of a particular use case.

2.1 A brief history of DNA data storage

2.1.1 Genes, proteins and nucleic acids

The idea that living organisms carry some information that is passed down from generation to generation originated in the nineteenth century. A Moravian monk named Gregor Mendel studied the inheritance of different traits in peas by interbreeding pea plants that showed different characteristics [14, p. 2]. He postulated the existence of hereditary factors, now called genes, which determined the different traits he observed. He published his work in 1866 [15], but it would not garner much attention until after his death.

In the early twentieth century, Mendel's work was picked up on and extended to form the field of genetics. At this point, it was not clear which element of the chromosome acted as the carrier of genetic information; the two contenders were proteins and nucleic acids. In 1944, Avery, MacLeod and McCarty, who were researchers at the Rockefeller Institute, published a paper [16] proving that nucleic acids were the information-carrying factor. They did so by studying the transformation of a non-virulent strain of bacteria to a virulent strain - they produced a cell-free extract from a virulent bacteria, which they divided up. They added various enzymes to the different extracts to break down each of the constituents of the cell in turn. They found that, when they broke down the cell's proteins or RNA, the transformation (from non-virulent to virulent) would still occur upon injection of the extract. But when

[1] With the exception of certain viruses that instead contain ribonucleic acid (RNA), which consists of a ribose sugar in place of a deoxyribose sugar [14, p. 3].

[2] RNA contains uracil in place of thymine; thus its four bases are abbreviated A, G, C and U.

[3] The sections of DNA which code for proteins. The non-coding sections of DNA are known as introns, and they serve a variety of functions which are still being studied.

[4] Two consecutive nucleotides can code for $4^2 = 16$ different amino acids, which is not enough. Three nucleotides can be used to code for $4^3 = 64$ amino acids, which is more than enough. Nature uses three-base codons but does not waste the remaining triplets - certain codons are used as markers for the start or end of a gene, and some codons map onto the same amino acids, allowing the genetic code to use one codon instead of another where it may be advantageous to do so.

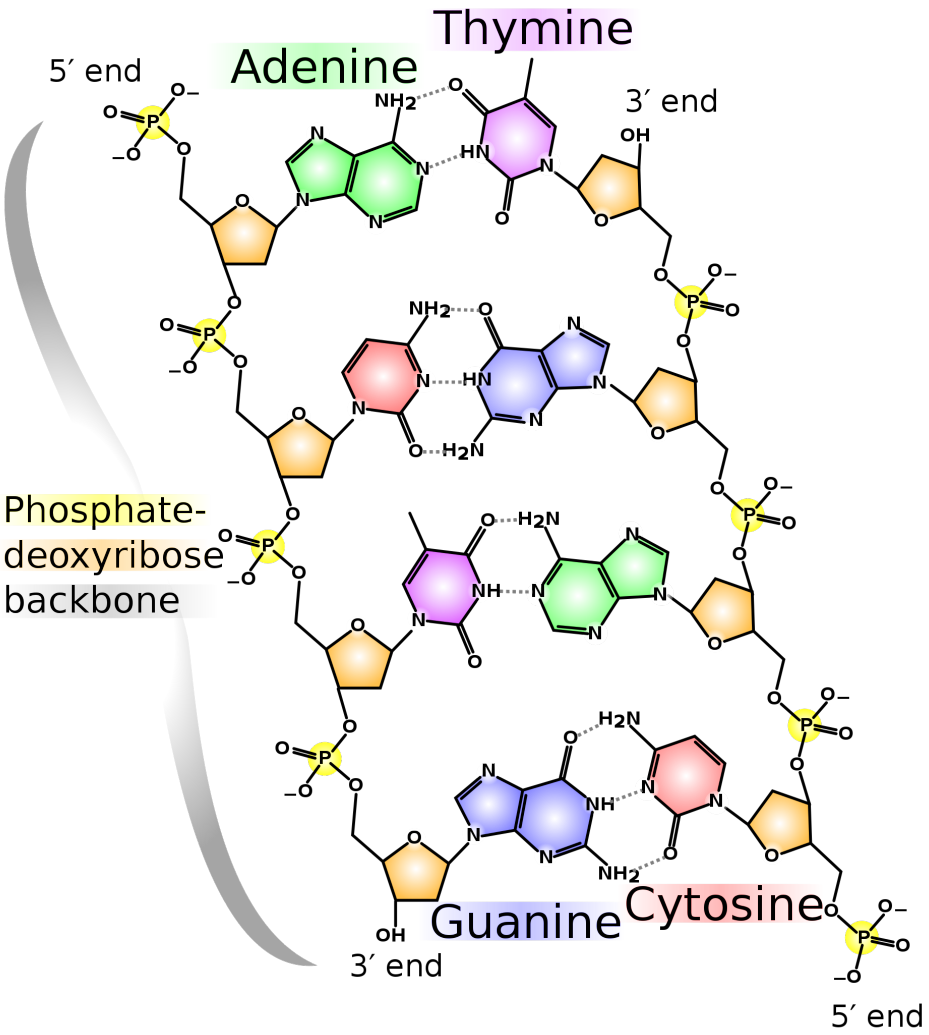


Figure 2.1: The structure of DNA. Illustration created by Madeleine Price Ball, provided freely via Wikimedia Commons, under the Creative Commons Attribution-Share Alike 3.0 Unported License. Available at: https://commons.wikimedia.org/wiki/File:DNA_chemical_structure_uk.svg.

---

they destroyed the DNA in the extract (using an enzyme called deoxyribonuclease [17, p. 250]) and subsequently injected it, no transformation occurred. Therefore it became clear that the carrier of genetic information was DNA, starting a race to deterimine its structure..

This race concluded in 1953, with the back-to-back publication of three seminal papers in volume 171 of Nature [18, 19, 20]. In the first of these papers, Francis Crick and James Watson presented a structure for DNA as a double-stranded molecule with bases on the inside. This model was based on X-ray crystallography data produced by Rosalind Franklin and passed on to Watson and Crick (indirectly through Max Perutz and Lawrence Bragg of Cambridge University) in the form of an informal report. Prior to seeing this data, Watson and Crick had hypothesised a three-stranded structure for DNA. The crystallography data, along with the famous "photo 51" (Figure 2.2) that had been taken by Franklin and Raymond Gosling, a PhD student, were published in the last paper of the three.

2.1.2 Nucleic acid computation

In *There's Plenty of Room at the Bottom*, Feynman states that,

> *The biological example of writing information on a small scale ... is not simply writing information; it is doing something about it.* [2]

He goes on to speak about creating our own 'very small things' which can be used to carry out tasks of our choosing. Feynman was speaking here about nanotechnology in a general sense, not necessarily about biological computers, but in 1994 Leonard Adleman showed that we could harness the 'tiny machines' provided by biology to solve computation problems. Adleman used DNA computation to solve an instance of the directed Hamiltonian path

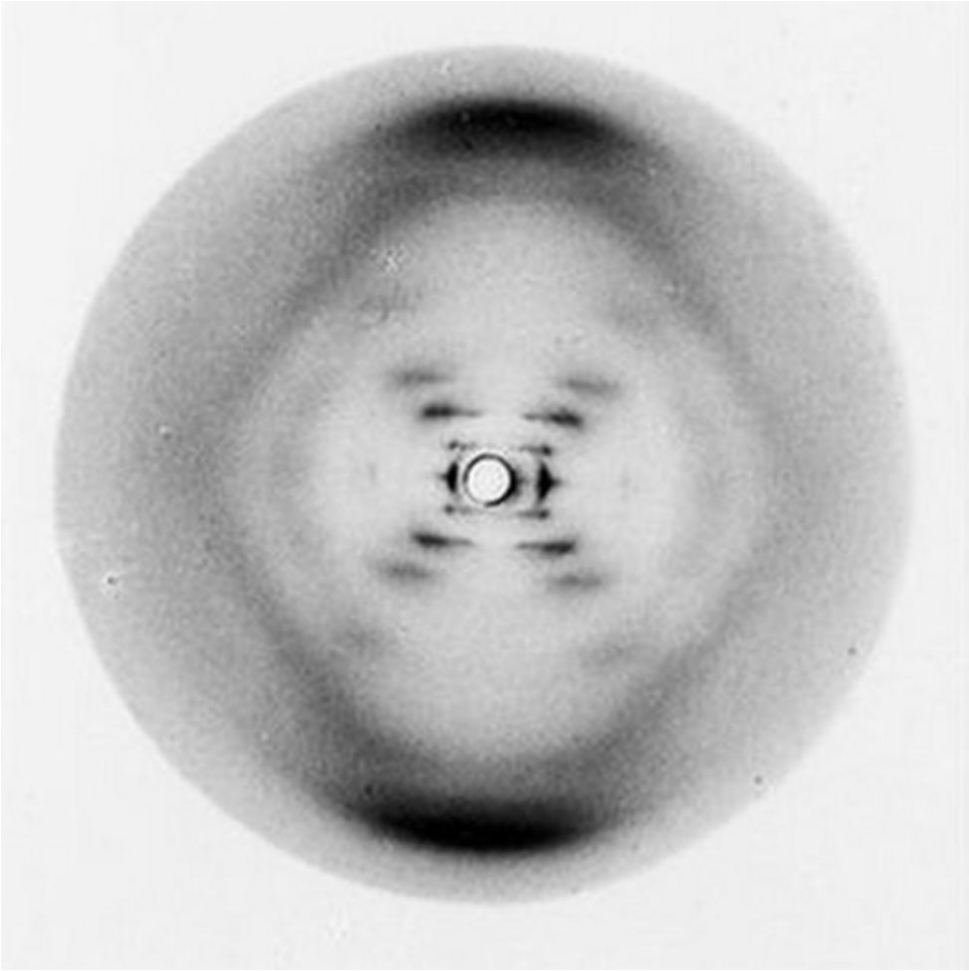


Figure 2.2: Photo 51, taken by Rosalind Franklin and Raymond Gosling at King's College London. [20]

problem [21], the aim of which is to find a path through a graph of nodes connected by directed edges, which visits every node exactly once (and no more). This problem is NP-complete, meaning it is computationally intractable[5] when tackled by conventional computers. Adleman demonstrated that a small version of the Hamiltonian path problem could be solved by synthesising strands of DNA to represent the nodes and edges of the graph and allowing these strands to join together into longer chains which represented a random path through the graph. He then used molecular processes to isolate any DNA strands which started and ended with the correct start and end nodes, had the correct length, and featured every node in the graph. If any such strand remained, then it must describe a Hamiltonian path through the graph. As described by Adleman, this algorithm essentially constitutes a 'massively parallel search' through a problem space. Adleman mentions that whilst a computation such as multiplication would be very difficult to do with a molecular computer (electronic processors can carry out such tasks very efficiently), certain intrinsically complex problems may be solved by taking advantage of the massive parallelism of DNA computing.

This work was extended in 1995 by Richard Lipton, who tried to provide a more general method to solve NP-complete problems [22]. He states that the brute force search method used by Adleman will not be sufficient to make biological computers advantageous since, despite the massive parallelism, the number of operations executed per second is too low compared to conventional machines. Lipton goes on to describe a scheme for solving the SAT[6] problem with a DNA algorithm. Lipton creates a technique for reducing SAT problems to a contact network with boolean expressions for edges (he states that other NP-complete problems can be mapped onto a contact network in a similar way). He then synthesises DNA strands to represent the nodes and edges of this contact network (which is reminiscent of the DNA graph constructed by Adleman), and uses various molecular operations to join the nodes together and extract strands of DNA matching a particular pattern. Through this algorithm, he is able to solve a SAT problem of N clauses using N DNA extraction steps. Lipton's main concern is whether DNA computers can actually be built to carry out these operations in a relatively error-free manner - researchers are doing exactly this type of work in the field of DNA computing.

The important result for the field of DNA storage, however, is the idea of synthesising arbitrary sequences of DNA and extracting them at a later date. The two papers by Adleman and Lipton provide inspiration for Eric Baum to

[5]The Hamiltonian path problem has exponential time complexity, meaning it takes exponentially longer to solve the problem as the size of the input, i.e. the graph, increases.

[6]The satisfiability problem - determining whether there are any variable values which will make a given boolean formula true. Much like the directed Hamiltonian path problem, this is NP-complete.

apply DNA to another computing problem - that of building a content-addressable memory.

2.1.3 Nucleic acid storage

Whilst Feynman expounded on the storage capabilities of DNA in his 1959 lecture, he never suggested that we may be able to make use of it as a storage medium. The first mention of the idea to use DNA for data storage came in the 1960s when American mathematician Norbert Wiener [23] and Russian scientist Mikhail Neiman [24] independently spoke about 'genetic memory' and suggested that it may have technological applications in the future. This speculation could go no further as DNA sequencing and synthesis technologies were only just being developed at the time. A practical example of DNA storage was first demonstrated in 1986 in a bio-art piece by artist Joe Davis titled 'Microvenus'. A 35-bit image of an ancient Germanic rune was encoded and synthesised into DNA and stored *in vivo*.

The first scheme to use DNA for storage purposes comes from Eric Baum, following on from the work of Adleman and Lipton mentioned above. In his 1995 paper, Baum describes a scheme for building an associative content-addressable memory[7] using DNA [25]. The method he describes is to encode a 'word' of information as a strand of DNA by defining a DNA subsequence to correspond to each bit of a word, i.e. a particular DNA subsequence would represent the first bit position of a word, another one would represent the second, and so on. If the subsequence is present in a strand, this implies that the word encoded by the strand has a one at the position represented by the particular subsequence. This means that the order of the subsequences in a strand is irrelevant, and it also means that a long word containing very few ones, for example, a sparse vector, could be encoded with a relatively short strand. Baum suggests that these stored sequences can be fetched with only partial knowledge of their content using magnetic beads. This could be done by creating a 'cue' (a short DNA sequence with a magnetic bead attached to it) for every bit position in a word - the cue would be the complement of the subsequence used to represent each bit position. One would then release each of these cues in turn, and they would bind onto DNA strands that contained the subsequences of interest. The magnetic bead could be used to extract these sequences. For instance, if we would like to search for all the stored words that have a one at positions two, three, and five, we would first use a cue for position two. This would result in us extracting all sequences that have a bit value of one at the second position. From this reduced pool, we would extract all sequences that have a one at position three and then again at position five. In this way, we could isolate all the sequences that match a pattern we are looking for and then sequence them to read the remaining content.

Baum mentions the possibility of creating a random-access memory with DNA by treating the 'content-addressed' subsequences as the address of a word and concatenating to this address a data sequence. He points out that the data sequence could be stored as a base-four word and thus would be very compact. In the case of an address, we are likely to have full rather than partial knowledge of it, so the process described above becomes easier - we simply need a single cue, which is the complement of the address we would like to read.

Baum's work was the first complete description of a DNA data storage mechanism but was entirely theoretical. The next practical step would come in 1999, when Clelland, Risca and Bancroft used DNA microdots to hide secret messages on paper [26]. This was the first example of DNA data storage that was carried out entirely *in vitro*, since Davis' work involved storing data within living cells.

A major breakthrough in the field of DNA storage occurred in 2012/13, when Church *et al.* [3] and Goldman *et al.* [11] independently decided to reconsider DNA for the purpose of archival storage. Both teams encoded around 650KB of data and stored it as DNA (using different encoding schemes) and were able to recover their data. Church *et al.* experienced 10-bit errors out of 5.27 million bits that they sequenced, which they state were predominantly due to homopolymers. Goldman *et al.* had no bit errors, but they did have two 25-nucleotide-long gaps in their sequence after reading it. They noticed that these gaps were part of a repeating self-reverse complementary section of DNA, which they hypothesized must have caused a sequencing issue. They used this repeating structure to estimate what the missing bases were (without referring back to their original encoded DNA sequences), and fill in the gaps, which enabled them to recover all the data they had stored with 100% accuracy. In the case of both teams, reading was done through PCR amplification followed by sequencing with an Illumina HiSeq sequencer. Church *et al.* state that they achieved a density of 5.5 petabits per mm^3, while Goldman *et al.* quote their density as roughly 2.2 petabits per gram, making the two approaches hard to compare as it is unclear how tightly the DNA is packed in either case.

Current synthesis technologies are unable to reliably synthesise long strands of DNA, so DNA is always synthesised in the form of oligonucleotide (short nucleotide sequences) of lengths up to 1000-nt[8]. For DNA storage purposes, it is common to use oligos of 100-300 nucleotides. Church *et al.* used oligos of length 159-nt, while Goldman *et al.* opted for 117-nt. They both made use of a large number of sequencing reads of each oligo to generate consensus for the value of every base position. Church *et al.* had an average coverage of around 3000-fold, the equivalent figure for Goldman *et al.* was 1,308.

An interesting difference between the approaches used by Church *et al.* and Goldman *et al.* was in the encodings that they utilised. Church *et al.* used a simpler encoding - they represented each bit with a single base (A or C for zero, G or T for one) and used this flexibility to partially mitigate homopolymers and extreme GC content, which are hard to read and write. They encoded their 5.27 megabits of data onto 54,898 oligos, each of which contained a 96-nt data block, a 19-nt address to allow them to reconstruct the data, and two flanking 22-nt sequences to aid with amplification and sequencing. Goldman *et al.*, on the other hand, first used a Huffman code to turn each byte into five or six base-3 digits, referred to as trits (a ternary digit, just as a bit is a binary digit). They then used this base-3

[7] A content-addressable memory is one in which a section of stored data can be accessed by partial knowledge of its contents.
[8] Oligonucleotide lengths are measured in the number of nucleotides, nt.

sequence to generate their string of nucleotides - each trit would be replaced by one of the three nucleotides which differed from the previous one written, thus ensuring that there were no homopolymers. The DNA sequences were split into overlapping segments to produce fourfold redundancy, with alternate segments being converted to their reverse complement. They added indexing information to each of their oligos, much like Church *et al.* and they also added a parity bit for error detection (but not correction). In total, their data was represented with 153,355 strings of length 117-nt. Goldman *et al.* also mention that only 10% of the synthesised DNA was used to reconstruct all of the data, meaning that the remainder could easily be stored for future reads.

At this point, the costs of synthesis and sequencing were far too high for DNA storage to be seriously considered for archival workloads. Goldman *et al.* estimated their costs to be $12,400 per MB for synthesis and storage and $220 per MB for sequencing. They projected that costs would have to come down by at least two orders of magnitude before DNA could be considered a viable option for sub-50-year archives.

2.1.4 Advancements: error-correction and random-access

Since Church *et al.* and Goldman *et al.* reignited interest in DNA storage, there have been a number of important steps forward. One of these has been in the form of improved error correction. The incorporation of ECCs is one of the main ways that the needs of DNA digital data storage differ from those of other fields, such as genetic testing. In most biological and medical applications, a strand of DNA must be synthesised or sequenced with no errors, 100% accuracy, in order to be of any use. When searching for a mutation in a gene, for example, a sequence with one or two errors in it is unacceptable; it is unclear whether there is a mutation or simply a sequencing error. Therefore, the primary focus for sequencing and synthesis technologies has been accuracy. In DNA storage, the sequences being synthesised are defined by the encoding used to create them - meaning that error-correcting codes, as have been used in computer science since the 1950s, can be incorporated into the DNA strand itself. Thus, after sequencing and decoding, the error-correcting codes can be applied to attempt to fix errors that have appeared.

This technique of incorporating ECCs is becoming increasingly important (every major attempt at DNA storage in the past 5-7 years has made use of error correction to some extent) as it is a primary tool to help push down the costs of synthesis and sequencing. Part of the reason that synthesis and sequencing are so expensive is the life sciences' need for accuracy. But since we are able to correct a certain proportion of errors in DNA storage, we can look for new synthesis and sequencing methods which sacrifice accuracy for increased speed and decreased cost. One example of a sequencing technique which comes with such a tradeoff is nanopore sequencing, especially with smaller, cheaper machines such as the MinION from Oxford Nanopore. Sequencing with the MinION is cheaper than, for example, using a large Next-generation sequencer, but it has higher error rates as well. This can be made up for with more advanced ECCs [27], and improved basecalling using neural networks [28].

The other major improvement that has been made over the schemes used by Church and Goldman is the capability for random access. This capability is not as mature and widespread as error correction, but it is likely to see an increasing focus in the coming years. In the papers published by Church *et al.* and Goldman *et al.*, and also in most other DNA storage schemes, data access occurs in a sequential fashion, meaning the entirety of the stored data is read in one go - all or nothing - there is no way to read only a part of the stored information. This is necessitated by the way the data is stored - synthesised strands of DNA are stored together, suspended freely, with no physical organisation like that of a magnetic drive/tape. To perform a sequential read, you simply sequence the available DNA until you have performed enough reads to reconstruct the data. In some cases, such as restoring a full set of data from a backup, a sequential read may be optimal. But in most cases, the desired information is only a small subsection of the data stored, meaning that a sequential read can be very wasteful in terms of cost and time since much of the read data will not be used. To avoid this wasteful access pattern, there are two primary techniques that could be used to achieve random access. The first is the magnetic bead extraction system proposed by Baum in his 1995 paper [25]. Magnetic bead extraction is used in the life sciences but has seen very limited use in DNA storage. An example of making use of magnetic beads for data retrieval is the work by Lin *et al.* [29]. A different magnet-based architecture can also be seen in the 'Magnetic DNA-based Random Access Memory' (MDRAM) system proposed by Lau *et al.* in an as yet unpublished paper from 2021 [30].

The far more common method for random access in DNA storage is PCR. During the encoding phase, unique primer sequences are assigned to different pieces of data and synthesised as part of the oligonucleotide for each DNA sequence. When a section of data needs to be accessed, the primers associated with that section are found and used to amplify the desired sequences of DNA. In this new pool of DNA, the desired sequences will be greatly over-represented, so they can be read by simply sequencing the pool. This technique was demonstrated to be effective by two papers independently in 2015 [31] and 2016 [32]. It was also later utilised by Organick *et al.* in 2018 to store over 200MB of data in DNA, the largest demonstration of DNA storage until that point [33].

2.2 The DNA data storage lifecycle

As can be seen from Figure 2.3, the major steps of the DNA storage lifecycle are writing, storing, retrieving, and reading. The writing step can be further subdivided into encoding and synthesis, while the reading step can be split up into sequencing, decoding, and correcting. A summary of each of these processes is provided below.

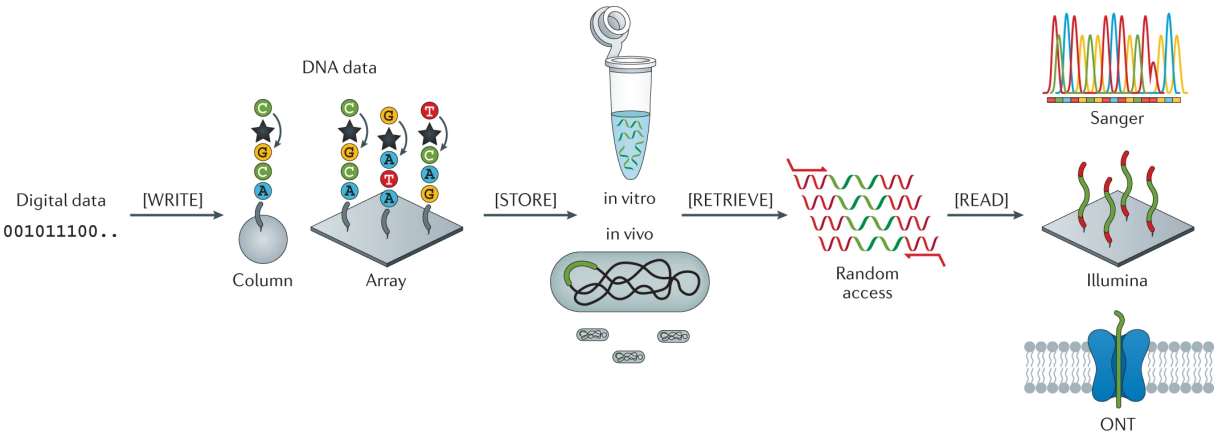


Figure 2.3: A demonstration of the DNA storage 'lifecycle'. Taken from Ceze *et al.*'s 2019 review of DNA data storage [6].

2.2.1 Information encoding

The encoding is the most computationally interesting part of the lifecycle. The fundamental idea behind it is to map a string of binary bits to a string of quaternary nucleotides. This mapping can be done in numerous ways, with each approach having a different profile of tradeoffs. As summarised by Heinis and Alnasir in ref. 34, the key factors in assessing different encodings are error correction, information density, data retrieval, and adherence to biological constraints. Data retrieval has already been covered in the section above - the two options are sequential and random access. The chosen method of data retrieval must be taken into account at the encoding stage since, if random access is to be achieved, primers must be added to the nucleotide sequences before synthesis.

A large number of encodings incorporate some form of error correction. This is crucial as synthesis and sequencing are both error-prone (insertions, deletions and substitutions are all relatively common), and during storage DNA strands may break. Therefore, redundancy is needed - synthesis operations naturally produce a great deal of physical redundancy (repeated copies of the same sequence), but DNA storage can be made a lot more fault tolerant through the incorporation of logical redundancy, i.e. ECCs. There are many different types of ECCs which have been designed and refined by the field of information theory - a small set of these are explored in section 2.3. ECCs are usually added along with indexing information at the first step of the encoding process before the mapping to nucleotides is carried out.

Clearly, the incorporation of ECCs comes at a cost - specifically in information density. A natural assumption regarding information density would be that it is a secondary concern in this field - as we have seen, DNA storage offers a density several orders of magnitude greater than other storage media, so it stands to reason that we would be happy sacrificing a large degree of it, even for small improvements in error rates. However, this is not the case. Information density is crucial due to the high cost of DNA synthesis (and also sequencing, but to a lesser extent); the more bits we can store using a single nucleotide, the fewer nucleotides we need to synthesise, therefore the lower the cost per byte. Information density may be measured in terms of volume or mass, but since in DNA storage, the fundamental unit of storage is a nucleotide, density is usually measured in bits per nucleotide. Traditionally, the theoretical maximum information density of DNA storage has been thought to be two bits per nucleotide. However, in 2018, two papers suggested using degenerate bases to increase information density beyond two bits per nucleotide. This can, in some sense, be thought of as reclaiming some of the information density lost to the vast parallelism of DNA storage. The idea is simple: along with our normal four bases, A, C, G, and T, we introduce a fifth symbol, A-T (or another combination of two nucleotides), which we can call a composite base. The reason we are able to do this is that we have a huge amount of parallelism - we have many oligonucleotides representing the same exact sequence. Therefore, when we sequence some stored DNA, we will sequence the same string over and over again, a fact which helps avoid certain errors. We can use this feature to create our A-T composite base - in half the oligos representing a certain base sequence, we include A at a particular position, and in the other half, we place T in that same position. When it comes to sequencing, it will become evident that this base position is split between A and T, and so it can be interpreted as a degenerate base, A-T. In this way, the alphabet has been expanded to include five different symbols, increasing the maximum achievable information density.

Whilst ECCs are very good for restoring corrupted data, we would still like to have as few errors as possible since fewer errors mean fewer bits reserved for error correction, which means higher information density and, therefore, lower costs. An encoding can lower the error rate produced during sequencing and synthesis by adhering to various biological constraints. The constraints that encodings often take into account include homopolymers, short tandem repeats, extreme GC content and secondary structure. The presence of any of these factors increases error rates in synthesis and sequencing, so many encodings will try to avoid them. For example, homopolymers and short tandem repeats increase the chance of PCR slippage, which can lead to deletion errors. Church *et al.* attempted to avoid homopolymers in their 2012 paper by using two different nucleotides to represent one and the other two to represent zero, allowing them to alternate between bases upon encountering consecutive repeats of the same bit. Goldman *et al.* avoided the same problem using a base-3 Huffman coding; thus, it can be seen that completely

different encoding mechanisms may be used to tackle the same problem.

2.2.2 Methods of synthesis

The two common techniques for DNA synthesis are the chemical method and the enzymatic method. The most established and widely used of the two is the chemical method, which is phosphoramidite-based oligonucleotide synthesis. This works through cyclical additions of reversibly 'blocked' mononucleotides to prevent the formation of homopolymers. In each cycle of the process, a single base is added - A, C, G or T. These mononucleotides have a 'blocking group' attached to them, which prevents two of the same base from being added at the same time, i.e. it prevents unwanted homopolymers. In the next step of the cycle, the blocking group is removed to allow the addition of the next base. The most common synthesis errors arise from issues in removing this blocking group [6]. This chemical synthesis method is amenable to parallelization. As synthesis itself is slow, greater throughput is dependent on greater parallelism, which can be achieved by increasing the number of spots that a DNA strand can bind to. This, in turn, is done either by reducing the size of the spots or by increasing the area of the solid substrate on which DNA is grown, allowing a greater number of spots to be placed on it.

The major alternative to phosphoramidite-based synthesis is enzymatic synthesis. This works through the use of DNA polymerizing enzymes, such as Terminal Deoxynucleotidyl Transferases (TdTs) (TdTs) [6]. These enzymes extend a DNA strand one base at a time in a process which has the potential to be faster than chemical synthesis while possessing the ability to create longer strands. This technology is not yet as mature as chemical synthesis but is likely to see increased development in the future.

A third possible mechanism for synthesising DNA is photolithographic synthesis. This is a newer synthesis method which is considerably cheaper than the other two and may have the potential to become faster as well. Its major drawback is its high error rate, which makes it unsuitable for most life sciences applications. However, through the use of advanced error correction and information reconstruction techniques, Antkowiak *et al.* demonstrated in a 2020 paper [35] that it could be used for DNA storage with perfect data recovery. This method is also likely to see an increased focus in the coming years.

2.2.3 Approaches to storage: *in vitro* or *in vivo*

Since the initial use of DNA to store hidden messages in microdots [26], the majority of DNA storage approaches have made use of *in vitro* storage. An advantage of this is that DNA can be stored more compactly when it is stored without the rest of the cell matter around it. Furthermore, *in vivo* storage may come with complications, such as errors being introduced during cell division and damage to the organism carrying the foreign DNA. That being said, the potential to use *in vivo* storage through synthetic biology is being explored [36, 37], as it is possible that it may provide solutions to some of the drawbacks of *in vitro* storage.

2.2.4 Retrieval: sequential or random access

In DNA storage and in computer science in general, there are two primary forms of data retrieval; sequential access and random access (as in random access memory or RAM). Sequential access is the simplest, as it is done by sequencing all DNA strands available and putting the retrieved data together using either indexing information or overlapping sequences. This means that, even if only a small subsection of data is desired, all the data in a given pool of DNA must be sequenced (and possibly reconstructed or re-amplified). This is, of course, slow and costly, though it requires minimal overhead. Therefore, for any application which does not necessitate the retrieval of small portions of data, sequential access should be considered. However, the majority of computing applications, even archival ones, benefit from random access, which is the ability to retrieve any section of data at will. In DNA storage, random access is a relatively new capability and is usually performed through the use of primers which flank the data-carrying segment of DNA (decreasing information density). This approach has been used in a number of large DNA storage experiments [31, 32, 33]. Another way to achieve random access is to make use of magnetic beads, as was proposed by Baum in his 1995 paper [25]. Magnetic approaches to data retrieval have been demonstrated in a number of studies [29, 30]. There are new approaches being developed for random access, such as making use of fluorescent oligonucleotide probes [38, 39]. It is very likely that random access will become the norm for DNA storage schemes in the future.

2.2.5 Methods of sequencing

The two primary types of DNA sequencing are sequencing by synthesis (SBS) and nanopore sequencing. SBS is a slow but high-accuracy method favoured by the life sciences. Its two most widely used forms are Sanger sequencing and Next-generation sequencing. Next-generation sequencing is also widely used for DNA storage and was the sequencing method used in the original papers by Church *et al.* and Goldman *et al.* It works by attaching single-stranded DNA molecules to a substrate surface, amplifying them by PCR, and annealing them to complementary bases with fluorescent markers. A laser directed towards the fluorescent bases allows a camera to process the different colours of light emitted by the different fluorescent bases, thus determining their sequence. A popular SBS sequencer is the Illumina HiSeq, which was used by both Church and Goldman. Disadvantages of SBS include the need for preparatory steps such as fragmentation and amplification [34].

The other major category of sequencing is nanopore sequencing. This has been gaining popularity in both DNA storage and the life sciences due to the fact that it is faster than SBS. Nanopore sequencing works by using a potential

difference (i.e. voltage) to direct negatively charged DNA molecules through a very small hole (a nanopore). This hole is part of an open circuit, and when DNA particles pass through the hole, the circuit becomes partially closed, which causes fluctuations in the electrical current passing through the circuit. These fluctuations allow bases to be differentiated. An advantage of nanopore sequencing over other methods is the real-time readout of data [6]. A further advantage is that the sequencing machinery can be made smaller, as demonstrated by sequencers such as Oxford Nanopore's MinION. This smaller, cheaper machinery comes at the cost of higher error rates, which are accounted for by more advanced ECCs and more powerful basecallers[9] that make use of machine learning [28, 27].

2.2.6 Data decoding

Once sequencing has been carried out, data decoding can begin. Each sequence of nucleotides will be mapped onto a string of bits in what is essentially the functional inverse of the encoding step. At this point, any incorporated ECCs can be applied, and indexing information can be used to place each chunk of data back into the correct position, thus reconstructing the stored data files.

2.3 Approaches for error-correction

2.3.1 Redundancy: physical and logical

All error correction is fundamentally based on the idea of including redundant data. The simplest form of this is physical redundancy, which means storing/sending several copies of the same piece of data. This way, even if some of the copies contain errors, it may be possible to recover the data by combining all the redundant information to achieve 'consensus'. This is essentially done by having each copy 'vote' for what the correct data is. And by including an odd number of redundant copies, you can ensure that a tie never occurs. This scheme for error correction relies on the absence of systemic errors - if every sequence of data has an error at a particular point, say, due to problems with sequencing, the consensus will converge on the erroneous data. Furthermore, physical redundancy is usually expensive; storing multiple copies of the same data is, for most storage technologies, wasteful. This is the primary advantage of logical redundancy; it allows data recovery whilst taking up much less space than simply having a spare copy. DNA storage is actually an exception to this - it is hard to avoid having physical redundancy due to the nature of the synthesis process. That being said, logical redundancy is still useful in DNA storage as it is better at dealing with systemic errors and allows files to be recovered with lower sequencing coverage.

In an attempt to send data accurately through a noisy transmission channel[10], the field of information theory was introduced by Claude Shannon in 1948 [40]. A full treatment of this topic is far beyond the scope of this report; the important conclusion with regard to DNA storage is that there is a mature and well-developed branch of computer science and mathematics which can be leveraged to ensure high-fidelity data storage in DNA. A few specific instances of logical redundancy are briefly mentioned below.

2.3.2 Parity checks and Hamming codes

The most basic form of logical redundancy is a parity check. This redundancy is sufficient only to detect errors, not to correct them. It works by checking the number of ones in a bit stream; it expects the number of ones to be even, hence 'parity'. The redundant element here is simply a single bit, often at the start or end of the bit stream. The redundant bit, or parity bit, is set by checking the parity of the data to be stored and, if the parity is odd, setting the redundant bit to a one, thus making the parity even. Or, if the parity is already even, the redundant bit is left as a zero. Upon receiving or reading the data, a parity check can be carried out. If the parity is odd, it is certain that at least one bit flip has occurred. If two bit flips occur (or any other even number of flips), the error is not detected, so this scheme is of limited capability. Goldman *et al.* added a parity bit to each of their oligos for the purpose of error detection.

This simple parity check led to the invention of the first error-correcting code by Richard Hamming in 1950 [41]. The idea behind it is relatively easy to understand; it works on the principle of maintaining a parity bit for every bit position in the index of the data to be sent/stored. For example, in order to store 256 bits, you would use eight parity bits, one for each bit position in the eight-bit address you would need to index into the 256 bits[11]. Each of these parity bits is set based on the same parity check idea described above, but only on a subsection of the stored data. For example, the first parity bit will be set by performing a parity check on all the bits whose 'bit address' contains a one in the first position. Then the second bit would be set via a parity check on all the bits whose address has a one in the second position, and so on. This scheme produces parity checks of overlapping sections of data. When the data is received/read, each parity bit is used to perform a parity check, if all of these are found to be even, no error is detected. Each parity bit is, in essence, indicating whether there was a bit flip in the region it is responsible for checking. Because these regions overlap, each parity check allows us to narrow down where an error might have occurred, eventually narrowing down to a single bit, which we can flip back to correct the error.

Hamming codes have the same limitation as parity checks (they can only correct a single bit flip), so are not as common as other more advanced codes. They do, however, provide the simplest example of the concept of using

[9] A basecaller is a program which estimates a sequence of nucleotides from a sequencing signal.

[10] Storage can be thought of as a data transmission channel; it is equivalent to sending data from the present to the future, and so all of the results concerning sending data through a noisy channel apply to storing data in an error-prone storage medium.

[11] With eight bits, you can index $2^8 = 256$ different 'addresses'. Normally, an address would store at least a whole byte of information, but in this case, we are trying to locate bit flips, so we need to be able to index every single bit.

logical redundancy to detect and correct errors, hence the provided explanation. More widely used codes are not as intuitive and make use of complicated mathematics - one such code is the Reed-Solomon code.

2.3.3 LDPC

Low-Density Parity Check codes are another form of error-correcting code based on parity checks. They work by defining a binary parity matrix which specifies a number of parity check equations. In this parity matrix, each row specifies a different equation, whilst each column specifies a different bit position. If, in the first row of the matrix, the first, second, fourth and sixth columns, this indicates that the parity of the first, second, fourth and sixth bit positions should be even. Each additional row of the parity matrix specifies a new parity equation which can be used to detect and correct errors.

2.3.4 Convolutional codes and Viterbi decoding

Convolutional codes are another simple form of error correction. They are the basis for the encoding scheme explored in this work. Convolutional codes work based on a FSM which starts in an initial state, and transitions to the next state based on the "input", i.e. the next bit to be encoded. Each time the code transitions to a new state, it also emits an "output" which is the encoded version of the input bit. Convolutional codes are defined in terms of their coding overhead, for example, a $1/2$ convolutional code encodes a two bit output for every one bit of input. Figure 2.4 diagrammatically represents a potential FSM for a $1/2$ convolutional code in a structure known as a trellis. In the trellis, the starting state is represented on the left, and the next state on the right. The blue arrows demonstrate the transition which occurs from each state if the next input is a zero, while the red arrows demonstrate the transition if the next input is a one. Each transition emits a two bit output which is the encoded version of the one bit input which caused the transition. For example, if the current state is 10, and the next input is a 1, the output 01 is emitted and the state 11 is transitioned to.

This form of encoding produces an encoded sequence which is longer than the raw sequence by a certain multiple - in the case of a $1/2$ code, the encoded sequence is twice as long as the original one. These encoded sequences are decoded using the Viterbi algorithm, which is an algorithm for estimating the maximum likelihood sequence, i.e. the raw sequence most likely to have produced an observed encoded sequence. The Viterbi algorithm is discussed in greater depth in section 4.2.5, which describes the implementation of Viterbi used in this work.

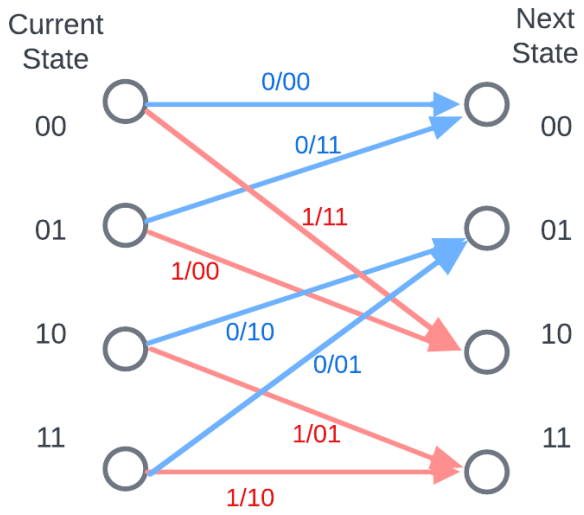


Figure 2.4: A trellis for a 1/2 convolutional code.

2.3.5 Reed-Solomon codes

Reed-Solomon codes are convolutional error-correcting codes which make use of advanced mathematics, such as Galois fields, to detect and correct errors. They work by splitting the data to be sent/stored into symbols and detecting erroneous symbols. Much like Hamming codes, the fundamental underpinning of Reed-Solomon codes is to introduce redundant information, which in some sense, summarises the information being stored. Unlike Hamming codes, there is no simple, intuitive understanding of Reed-Solomon codes, and the mathematics involved is beyond the scope of this report. The fundamental feature of this error-correcting scheme is that, unlike Hamming codes, it is able to correct multiple errors, even when they occur next to each other in a short burst. They are also able to deal with a wider range of error types - Hamming codes could deal with substitution errors, but deletion and insertion are also common in DNA storage. Reed-Solomon codes are able to detect and correct these errors (to an extent). These advantages have resulted in Reed-Solomon codes seeing widespread usage in various DNA storage schemes [6].

2.3.6 Inner and outer codes

The previous two ECCs described were both examples of inner codes, which are codes placed within a DNA strand in order to correct errors that occur in that same strand. There is another style of error correction (often used in conjunction with inner codes), which works by adding whole new strands of logically redundant information, i.e., a whole strand that is dedicated to error correction. This allows the detection of a different kind of error that inner codes cannot deal with - the erasure of a whole strand. If a whole strand is not sequenced, perhaps because it was not synthesised or because it became massively underrepresented during PCR amplification, inner codes will not help since these are placed on the strands that have gone missing. Therefore, outer codes placed on their own strands may help detect that a strand has gone missing or even reconstruct the missing strand.

2.4 Future potential

Several attempts have been made to assess the future potential of DNA storage, these include the work by Heckel *et al.* in 2017 [42] and 2019 [43], and also the commentary on nucleic acid memory by Zhirnov *et al.* in 2016 [4]. Zhirnov *et al.* make an interesting projection in terms of the world's silicon supply - they point out that the predicted global memory supply in 2040 of 3×10^{24} bits will exceed the estimated silicon supply of the world and imply that DNA could help provide an alternative basis for storage. They point out that DNA has a volumetric density three orders of magnitude greater than flash memory and an energy of operation eight orders of magnitude less. It is not a perfect comparison since, at this point in time, it is very hard to imagine DNA being used for short-term memory instead of very long-term storage, so the requirements for RAM or SSDs are unlikely to be fulfilled by DNA.

However, they go on to perform an interesting analysis using an energy-barrier model - a model in which information is described as a resource protected by an "energy barrier", which is a barrier that requires a certain amount of energy to break through. Such models can be used to calculate the information retention rates of different storage media over time. They analyse memory consisting of different information-bearing particles, such as electrons (flash memory), magnetic domains (HDDs[12]), and molecular fragments, as in the case of nucleotides in DNA. There are two primary types of unintended energy transitions which can allow information-bearing particles to get past an energy barrier and cause information loss. The first is a classical transition (thermally excited over-barrier transitions), and the second is a quantum transition (through-barrier quantum tunnelling).

Zhirnov *et al.* point out that with electron-based memory, quantum transitions are the dominant factor due to the size of the particles. Molecular fragments are larger and heavier, so quantum transitions are 'suppressed'. The authors claim that this leads to the potential for much greater storage densities. According to the calculations put forward by Zhirnov *et al.*, quantum transitions limit electron-based memory to a density of one bit per $\sim$10-15 nm, in comparison to a bit size of $\sim$0.3nm in DNA.

The suggestion being made here is important, as it does not simply claim that DNA is denser than our current electronic media (which is already well-known) but rather implies that DNA is denser than our electronic media can possibly be, even in the limit of their density.

[12] Hard disk drives.

Chapter 3

Related Work

DNA archival storage, a bottom-up approach

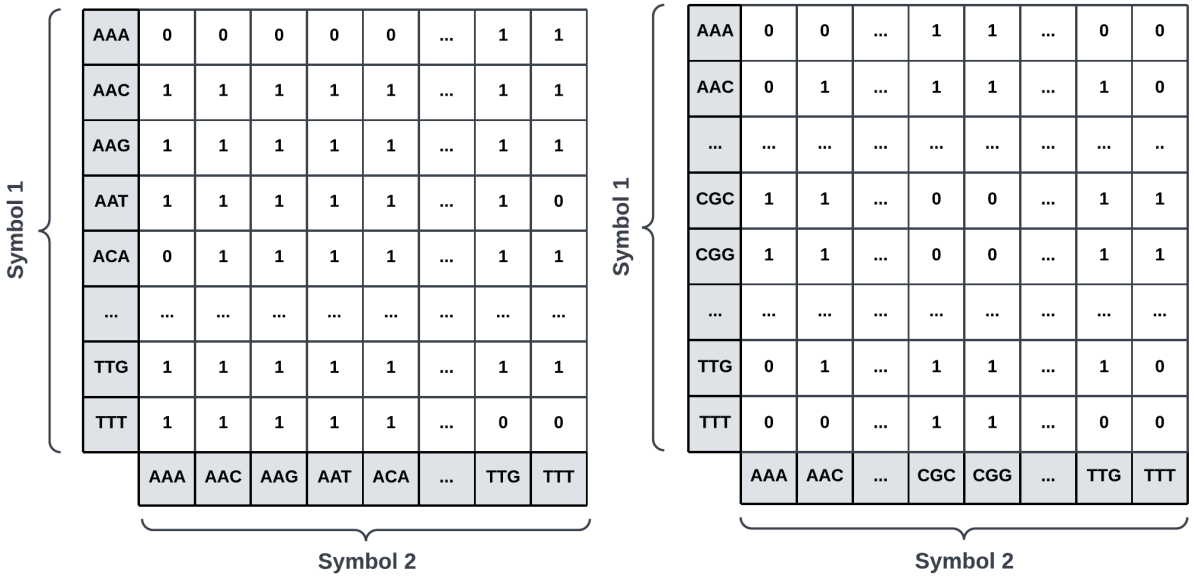
This thesis is a continuation of the work carried out by Sella *et al.* in [13]. The encoding scheme analysed and extended by this project was first presented by Sella *et al.* as a means of generating encodings based on a set of constraints. The rationale for creating such an encoding scheme is centred around the idea that as synthesis and sequencing technologies are innovated upon, the set of constraints that an encoding must take into account is likely to change; thus, an encoding targeting a prior set of constraints may become obsolete. In order to address this issue, the researchers from Cambridge University and Imperial College present what they describe as "a recipe for taking constraints and producing an appropriate encoding scheme". They explain that such a mechanism will allow the DNA encoding to be moved in lockstep with technological advances. Central to this "recipe" is the process of taking a set of constraints and producing an FSM which describes how a DNA sequence should be produced from raw data. It is crucial to understand that this FSM fully describes the encoding generated by the "recipe"; once the FSM has been defined, standard algorithms for convolutional coding and Viterbi decoding can be used to encode and decode the DNA.

The FSM is constructed using the following process. First, the symbol length, L, is defined. This can be thought of as the length of a "word" of encoded DNA. The aim is to find a sequence of DNA symbols of this length which does not violate any of the constraints in the constraint list. To this end, a simple assumption is employed; that none of the individual symbols of length L contains a violation of the constraints[1]. If the assumption holds, one can be assured that an individual symbol will not on its own contain a homopolymer or reserved subsequence violation, and so only the concatenations of these symbols need to be considered when checking for violations. To ensure that this assumption is true, it is necessary to set the symbol size to a value that is, firstly, less than or equal to the maximum length homopolymer permitted, and, secondly, strictly less than the length of the shortest reserved subsequence. Once an appropriate symbol size has been selected, the process of constructing the FSM can begin. The following algorithm is used; every possible concatenation of two symbols is considered, forming a connectivity matrix, with a row for each possible value of symbol one and a column for every possible value of symbol two. This can be thought of as a transition matrix in which symbol one represents the current state and symbol two represents the next state. Each of these possibilities is checked against the entire list of constraints, and the matrix is populated with a one for every valid concatenation and a zero for every invalid (i.e. constraint-violating) one. Figure 3.1 shows sections of two possible connectivity matrices, each with a symbol length of three and only a single constraint to satisfy.

The algorithm proceeds to allocate a number of reserved bits (one at a time) at the front of the second symbol. It does this by "halving" the connectivity matrix; the matrix is essentially split into two by a vertical line, and the two halves are stacked on top of one another by taking an elementwise maximum. Each time this halving operation is performed, one bit of information (half a nucleotide) is relinquished as a reserved bit. Taking as an example the first matrix from figure 3.1, this stacking operation would cause the entry at (AAA, AAA) to be stacked on top of the entry at (AAA, GAA), since GAA is the column just after the halfway point of the matrix. This would mean that whether the first nucleotide is an A or a G is no longer dependent on the input data, rather that bit of information is reserved for the encoding mechanism (since A=00 and G=10, the second bit is deterministic, only the first bit is reserved). Assuming our only constraint is a maximum homopolymer length of four, the concatenation of AAA and AAA is invalid (the value in the (AAA, AAA) would have been zero), but the concatenation of AAA and GAA is valid (the value in (AAA, GAA) would have been one) - therefore the elementwise maximum of these two squares is one. In binary terms, this means that if the first symbol is 000000 (AAA), and the next input to append is 00000 (0AA), there is at least one reserved bit candidate that can make this concatenation valid; in this case, the reserved bit must be set to one to avoid a homopolymer. Therefore the concatenation of the symbols will be 000000100000 or AAAGAA.

If, however, the constraint is to have a GC content between 30% and 60%, neither (AAA, AAA) nor (AAA, GAA) are valid, so both of their values will be zero, meaning that the elementwise maximum from the halving operation will also yield a zero. In this case, another halving operation must be performed. This halving will place the cell representing the combination of (AAA, AAA) and (AAA, GAA) on top of the cell formed from the maximum of (AAA, CAA) and (AAA, TAA). This corresponds to having two reserved bits since the encoding mechanism now controls

[1] Strictly speaking, many of the symbols may violate the GC limits constraint, however, unlike the constraints against homopolymers and restricted sequences, a violation of the GC limits constraint may be fixed by concatenating two symbols together.



(a) Constraint: maximum homopolymer length of three.

(b) Constraint: GC-content between 30% and 70%

Figure 3.1: Two connectivity matrices with symbol length three. A one indicates that the concatenation of symbol 2 to symbol 1 does not violate the constraint, and a zero indicates that the constraint is violated.

whether AAA is followed by AAA, CAA, GAA or TAA. However, this is still not enough; none of these concatenations has a GC content higher than 30%. Therefore, one more halving operation must be performed, which puts the combined cell representing the maximum of the cells (AAA, [AAA, CAA, GAA, TAA]) on top of another combined cell holding the maximum of cells (AAA, [AGA, CGA, GGA, TGA]). Since AAA-CGA and AAA-GGA both constitute valid combinations (their GC contents are 33%, greater than the lower bound of 30%), the new cell has an elementwise maximum of one. In order to achieve this, three bits had to be relinquished out of six total bits, so the encoding has a 100% coding overhead, much like a $1/2$ convolutional code. In terms of encoding data, this means that if the last symbol to have been written was AAA (000000), and the next three bits to write are 000, either CGA (011000) or GGA (101000) can be appended as the next symbol. Hence, there is a choice between two "candidates" for the values of the reserved bits; 011 or 101. In their paper, Sella *et al.* do not specify any methods for choosing between these options, so in this work, several alternatives are proposed and compared.

While concluding, Sella *et al.* mention several possible directions for further work. They indicate that their use of an FSM in this encoding is intended to allow it to be augmented to an ECC. They also mention that "further work includes a choice mechanism for the values of the reserved bits". In this work, both these directions are pursued; a number of choice mechanisms are proposed and implemented, and their effect on the efficacy of the encoding as an ECC is assessed.

Chapter 4

Error-correcting encoder implementation

4.1 Encoding Scheme Overview

In order to assess the effectiveness of the encoding scheme from [13] in generating encodings with error-correcting capabilities, an implementation of it is required. Initially, this implementation was built entirely in Python for the purpose of development speed, however, due to performance issues, several key sections were rewritten in Rust and packaged into libraries. The resulting performance improvements are profiled in section 4.3.

An important departure of the work done in [13] is the way in which the number of reserved bits is defined. The method proposed by Sella *et al.* involves "halving" the connectivity matrix (i.e. incrementing the number of reserved bits) until there are no longer any constraint violations, as described in chapter 3. In this work, an area of interest is how the error-correcting capability of the produced encodings changes as the number of reserved bits is varied. Therefore, the number of reserved bits is defined as a separate parameter, which is controlled directly. If the number of reserved bits selected is insufficient to meet the given constraints, an error is simply returned. The advantages of this approach are that experiments can vary the numbers of reserved bits without performing the halving process multiple times to generate each FSM and that the constraint list and the number of reserved bits can be varied independently of each other so that their effects on error correction can be disentangled.

4.2 Project Design

As the purpose of this implementation is to run experiments and not to be used in production, the initial approach followed was to work fast and not worry about technical debt. However, some attention was still required to properly lay out the various submodules to allow for easily changing the parameters and design of the experiments. Generating an encoding relies on a number of processes shown in figure 4.1, starting with a way to represent the biochemical constraints, a choice mechanism for the reserved bits, a representation of the resulting FSM, a convolutional coder and a Viterbi decoder. As the produced data is to be synthesised into DNA, functions mapping bits to nucleotides are also required. Finally, in order to assess the encodings efficacy as an ECC, code is needed to inject errors and run experiments. The various submodules are expanded upon below, while the experimental design, methods and results are the subject of chapter 5.

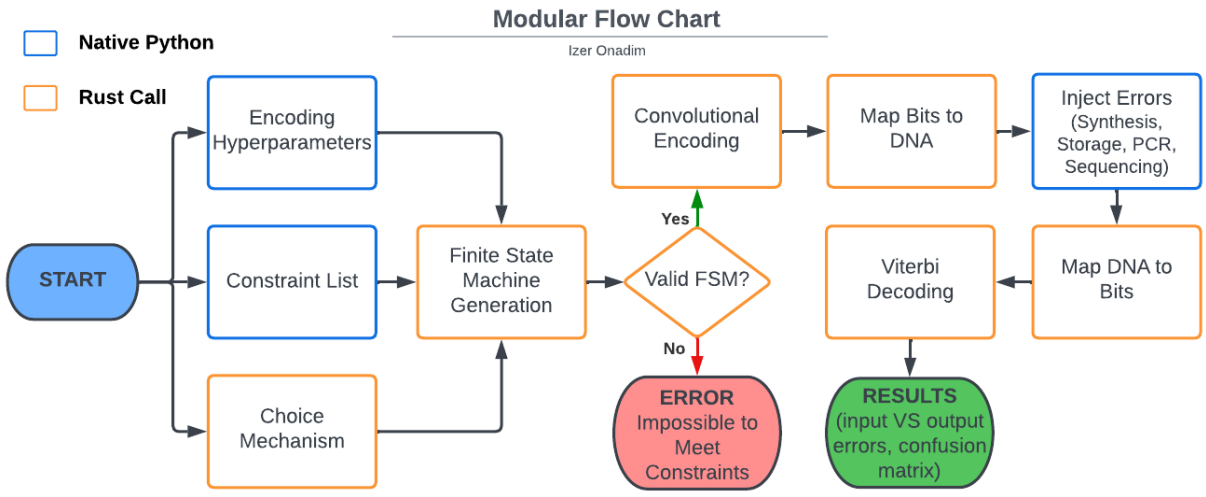


Figure 4.1: A flow diagram of the encoding scheme implementation. Each process corresponds to a separate submodule. A key difference with the encoding described by Sella *et al.* is inclusion of reserved bits as a hyperparameter. All modules were initially written in Python. The colour coding shows which modules were later rewritten in Rust for better performance.

4.2.1 DNA to Bit mapping

The mapping of bits to nucleotides is the simplest portion of the encoding. In essence, it constitutes a complete encoding by itself. However, if this mapping was the entire encoding, there would be no protection against errors or biochemical constraints. A common mapping of bits to nucleotides is shown in table 4.1, which is also the one used in this work.

A	C	G	T
00	01	10	11

Table 4.1: A common mapping of bits to nucleotides, used in this work.

Whilst simple (only 4! = 24 possibilities), the mapping of bits to DNA is not irrelevant. One reason that not all mappings may be equally effective is that not all types of errors are equally likely. For example, take the confusion matrix in table 4.2.

	A	C	G	T
A	100	0	0	0
C	0	95	5	0
G	0	5	95	0
T	0	0	0	100

Table 4.2: A potential confusion matrix of DNA synthesis/sequencing errors[1].

This matrix demonstrates a case in which nucleotides A and T are synthesised and sequenced with 100% accuracy, however, nucleotides G and C are confused with one another at a rate of 5%. In this case, assuming an equal distribution of nucleotides[2], it can be seen that the overall nucleotide error rate would be 2.5%. If the mapping in table 4.1 is used, this will also translate to a bit error rate of 2.5% since the bit representations of G and C differ by two bits (i.e. have a hamming distance of two).

A	C	G	T
00	01	11	10

Table 4.3: An alternative mapping of bits to nucleotides.

If this pattern is observed, the bit error rate could be halved by simply making use of the alternative mapping in table 4.3, in which the bit representations of G and T are switched. The result of this change will be that G and C differ only in the first bit of their representation (i.e. the hamming distance has decreased to one), therefore, each nucleotide error will translate to a single bit error rather than two. This will halve the bit error rate from 2.5% to 1.25% without changing the error profile of the storage channel at all.

In this work, the assumption has been made that the error rate across different nucleotides is roughly the same. Whilst this is not necessarily true, it is sufficient to test the error-correcting capabilities of the encoding. Hence, in this case, the choice of mapping is arbitrary and should have no impact on the encoding's performance.

4.2.2 Representing Constraints

There are a number of biochemical 'constraints' that an encoding may be designed to obey. In reality, these constraints are not stipulated by chemistry, rather they are often required by companies which carry out DNA synthesis to avoid certain types of sequences which are harder to synthesise or more likely to produce errors. The encoding scheme attempts to account for these constraints when generating an encoding. The most common constraints applied by DNA synthesis companies are homopolymers, short-tandem repeats (STRs), GC limits, GC variation and reserved subsequences. Out of these, the encoding mechanism presented in [13] successfully complies with the constraints on homopolymers, GC limits and reserved subsequences. This work extends the encoding scheme to additionally deal with GC variation through a selection of choice mechanisms discussed in sections 4.2.4 and 6.1.1. This work also extends the encoding scheme to partially mitigate STRs through a novel mechanism described in section 6.3.

[1] This is a highly simplified error model; neither an error rate as low as 0% for A and T nor an error rate as high as 5% for G and C are realistic.

[2] The assumption of a roughly equal distribution of nucleotides is not necessarily unrealistic, however, it should never be relied upon since there are no guarantees about the data a user may want to store. In this case, the assumption is useful in simplifying the argument.

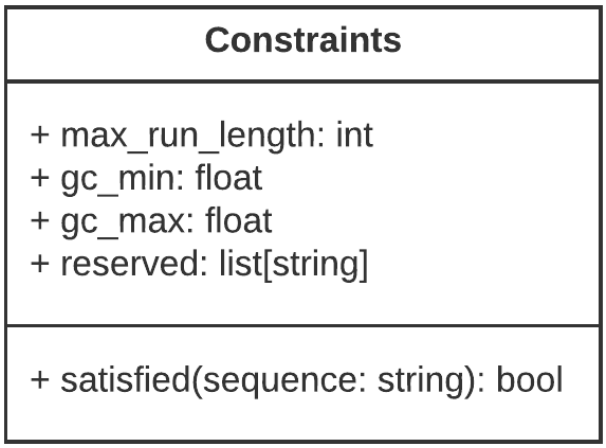


Figure 4.2: An object diagram demonstrating how constraints are represented in the implementation. The actual implementation can be found in appendix A.6.1. The `satisfied` method returns true if the given sequence does not violate any of the constraints.

Homopolymers are contiguous repetitions of the same base. This is a very common constraint, usually specified in terms of a "maximum run length", i.e. the maximum number of contiguous repetitions of the same base permitted. The `Constraints` type takes this parameter explicitly. The `satisfied` method of `Constraints` will return false if the given subsequence contains a homopolymer which exceeds this length. The maximum run length must be at least as long as the symbol length used for the encoding.

Short-Tandem Repeats (STRs) can be thought of as a generalisation of homopolymers. They refer to any repeated subsequence. For example, "ACGTACGT" is an STR as the subsequence "ACGT" appears twice consecutively. There is no explicit handling of STRs in the original description of the encoding scheme by Sella *et al.*, however, this work proposes a novel mechanism for partially mitigating violations of this constraint in section 6.3.

GC Limits are upper and lower bounds placed on the GC content of a DNA sequence. The GC content is simply the proportion of a sequence which consists of Gs and Cs. For example, the sequences "ACGT", "CGCG" and "ATAT" have GC contents of 50%, 100% and 0%, respectively. GC limits are taken into account in this work through the `gc_min` and `gc_max` attributes of `Constraints`. Setting a `gc_min` to 0.25 and a `gc_max` to 0.75 will result in `satisfied` returning false for any sequences with a GC content less than 25% or greater than 75%.

GC Variation is a limit on how much GC content can vary between two subsequences of a sequence. For example, a DNA manufacturer may stipulate that the GC content difference between the highest and lowest GC content subsequences of length 50nt[3] should be no more than 40%. This would mean that, in a particular DNA sequence, if the lowest GC content in a subsequence of length 50nt was 25%, the highest GC content in a different subsequence of the same length should be at most 65%. This factor is not explicitly controlled by the `Constraints` type, however, this work provides two ways of complying with GC variation constraints. The first is to make the difference between the `gc_min` and `gc_max` values less than or equal to the maximum allowed GC variation, and use a symbol length that is far shorter than the window size used for measuring GC variation (in the above example the window size was 50nt). Though Sella *et al.* did not point this out in their work, selecting such values for the GC limits means that their encoding scheme is able to decrease GC variation, however, violations are still possible if a short enough window size is selected. The second is to make use of one of the GC-Tracking choice mechanisms contributed by this project and described in sections 4.2.4 and 6.1.1.

Reserved Subsequences are subsequences of nucleotides which should not appear in the output of the encoding. These subsequences may be reserved for a number of reasons, the common ones being that they are used as primers in PCR for sequencing or synthesis, or that they are used as indexing sequences in a random-access DNA system. The `Constraints` class has the attribute `reserved` for reserved subsequences. Each reserved string in this list will be checked against any sequence given to `satisfied`; if the sequence contains one of the reserved subsequences, satisfied will return false.

This constraint list is defined by the user, though a reasonable default is provided, and used in the construction of an FSM which represents the encoding.

4.2.3 Finite State Machine

The FSM is the central aspect of the encoding scheme. It contains all the information necessary to perform an encoding or decoding operation on a bit string. It is simply a tuple of four attributes, which together can be used to encode a sequence with convolutional codes or decode it with the Viterbi algorithm. These four attributes are the input size, the output size, the initial state and the transition table. The purpose of each of these attributes is explained below.

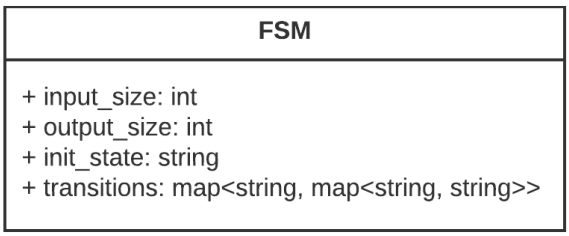


Figure 4.3: The structure of the FSM object/tuple. These four attributes are all necessary and sufficient for performing the encoding and decoding operations on a sequence.

input_size represents the number of data bits[4] to be included in each symbol. For example, if an encoding is to be generated with a symbol length of four nucleotides (meaning a bit symbol length of eight), and three reserved bits are required to meet the given constraints, the input size will be five bits since this is the number of bits remaining which can be used for data storage.

[3] Nucleotides.
[4] As opposed to reserved bits.

output_size is identical to the symbol length, except measured in bits rather than nucleotides. Therefore, if the symbol length is four, the `output_size` will be eight. This is referred to as `output_size` since it is the size of the output of the FSM, but it is also the size, in bits, of each FSM state.

init_state is the starting state which will be used for both the encoding and decoding. This needs to be included in the FSM as the same starting state must be used to decode a sequence as was used to encode it. For all experiments in this work, the starting state used is all zeros (i.e. a string of length `output_size` filled with zeros).

transitions contains the transition table, which describes how pairs of current state and input map onto the next state and output. A feature of this encoding generation scheme is that the output and next state are the same, thus storing both of these values is completely redundant. This is done since the FSM generation module is designed to be decoupled from the convolutional/Viterbi module, which allows the convolutional coder and Viterbi decoder implementations to be used to encode/decode any convolutional code. This decoupling was useful for testing each module separately, but also for comparing the encoding generator against standard convolutional codes such as the 1/2 and 2/3 codes. The results of these comparisons can be seen in section 7.2.

Generating the FSM

The input and output size and the initial state are parameters to be decided upon by the user of the encoding generator. The user also selects the constraint set and choice mechanism, which are used to construct the transition table. The role of the choice mechanism is covered in more detail in 4.2.4. A simple but novel algorithm is then used to construct the transition table (the implementation of this algorithm has been omitted for brevity but can be seen in appendix A.1). The idea behind this algorithm is simple; based on the input and output sizes given, it generates every possible state (the state is the previous output of the FSM) and the input string. For every pairing of these strings, it filters through all the possible choices for the values of the reserved bits. Each valid choice for the reserved bit values is called a "candidate". Next, the choice mechanism is given a set of candidates, as well as the input/output pairing they are associated with, and returns a single "chosen" candidate. A visual representation of this process can be found in figure 4.4. Transition table construction relies on the `satisfied` method of the `Constraints` object (to filter the reserved bit options by their validity) and the bits-to-DNA mapping which are described above.

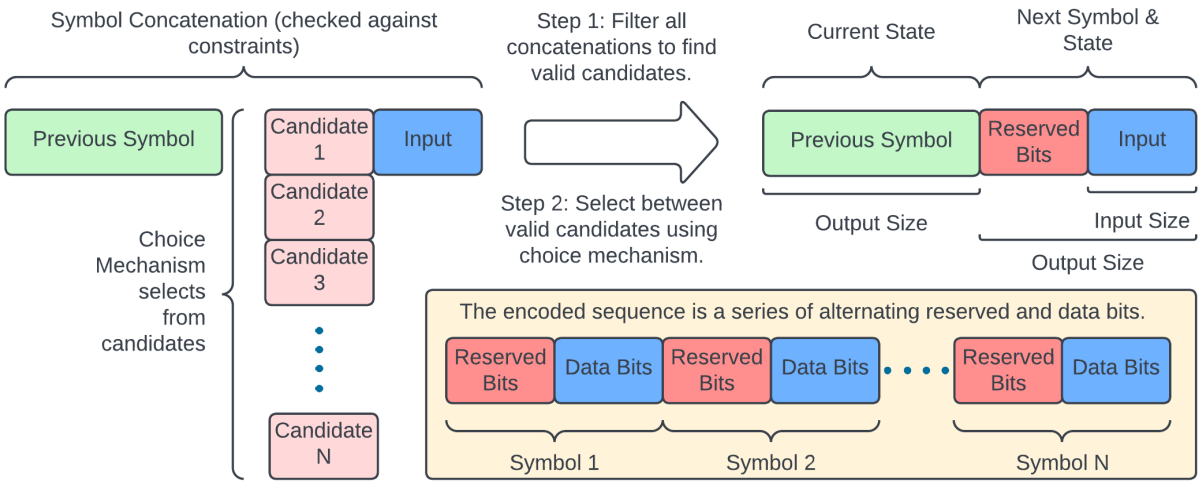


Figure 4.4: The process for selecting the reserved bits for each pair of state and input. The `Constraints.satisfied()` method is used to filter through the options and find the valid candidates. The choice mechanism is then used to select one of the candidates. A sequence is constructed by repeatedly appending new symbols based on the current state and input, so it will consist of an alternating series of reserved and data bits.

4.2.4 Reserved Bit Choice Mechanism

The mechanism for choosing between the different reserved bit candidates was left as further work in [13]. The choice mechanism is a function which takes as arguments the current state (i.e. the last symbol appended to the DNA sequence), the input (the next bits to be written) and the candidates (a list of valid possibilities for the values of the reserved bits), and returns a single chosen reserved bit string. In his Ph.D. dissertation [44], Omer Sella suggests two possible mechanisms, which this project implements and assesses with respect to error correction. This work also goes on to suggest several other choice mechanisms and assess their effect on error correction (chapter 6). The two mechanisms suggested by Dr. Sella are random choice and GC-tracking, both of which are elaborated upon below.

Random

The simplest mechanism is simply to pick randomly between all the valid candidates. The random choice mechanism comes with clear advantages in terms of simplicity, speed and also, as shall be shown in chapter 5, error correction. The implementation of this is simply a random choice from a list.

GC-Tracking

Another option for the choice mechanism is GC-tracking. This choice mechanism will decrease the GC variance of the produced DNA sequence, thus extending the encoding to meet another common constraint applied by DNA manufacturers. It is performed by finding the reserved bit candidate that brings the GC content closest to the target value of 50%. Crucially, if there are multiple values of reserved bits which have the best achievable GC content, it will simply pick the first one it found. If it finds a candidate which brings the GC content of the concatenation site to the optimal value (50%), it will stop searching through the other candidates. The implementation of this choice mechanism can be seen in appendix A.4.1.

4.2.5 Convolutional Codes and Viterbi

Encoding

Once the FSM has been constructed, the convolutional coder can use it to encode a binary sequence. Starting at the `init_state`, the convolutional coder advances in steps of `input_size`, and uses the FSM to determine the output (with length `output_size`) and the next state at each step. This produces a new encoded binary string, which is longer than the original message by a factor of `output_size / input_size`[5]. This encoded string is then translated into nucleotides using the DNA mapping as discussed in section 4.2.1.

Decoding

The first step of decoding is to apply the inverse DNA mapping to translate the nucleotide sequence back into a bit string. After this, the bit string is passed to the Viterbi algorithm along with the FSM that was used to encode it. There are many implementations of the Viterbi algorithm in different languages. Rather than using one of these existing implementations, this project constructs its own implementation, to allow for further modification and extension specific to its aims. Some of the modifications attempted are detailed in chapter 6.

Viterbi is an algorithm which, given a sequence of observations, finds a sequence of underlying states with the highest likelihood of having produced the observations. This makes it useful for decoding convolutional codes, where the sequence of observations is an encoded message received over a transmission channel or read from storage, and the underlying sequence of states is the estimate for the original raw message from which the encoded message was produced. The Viterbi algorithm does this by traversing the sequence of observations, and for each state transition which could have resulted in an output symbol, assigning a cost based on the hamming distance between the observed symbol and the output which would have been emitted if that transition had taken place. These costs are accumulated as the algorithm progresses through the observed sequence.

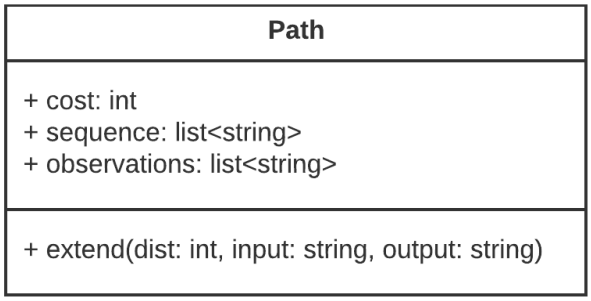


Figure 4.5: The Path object used by the Viterbi algorithm.

The implementation of Viterbi used in this work relies upon the notion of a `Path`. This object contains a cost, a sequence and a list of observations. The cost is the total accumulated cost of all the transitions along a possible path, assuming that the list of observations was produced by the sequence in the path. As the algorithm progresses through the observed sequence, it maintains a hash map which maps each state to the lowest cost path of symbols terminating at that state. Once it reaches the end of the observations, the lowest cost path out of all the paths in the hash map represents the maximum likelihood estimate for the sequence which produced the encoded message. The implementations of convolutional coding and Viterbi decoding are not included here as these are established algorithms, however, they can be found in appendix A.

4.2.6 Methods of Error Injection

In order to test the efficacy of the encodings generated by the generation scheme, it is crucial to inject errors into the DNA sequence between the steps of encoding and decoding. The aim in doing this is to simulate the errors which may occur during the synthesis, PCR, storage and sequencing processes. The initial approach followed in this project is to focus on substitution errors, which are the most common and easiest to address. Deletions and insertions can be detected using the length of an oligonucleotide, and so one (albeit expensive) approach for dealing with insertions and deletions is to simply ignore all sequences of the wrong length. This can be done due to the massively parallel nature of DNA synthesis - when synthesising an oligo, a large number of copies are synthesised at the same time. PCR is also cheap and fast, so this can always be applied before sequencing to ensure that there are many copies of the same oligo to be sequenced.

When injecting errors, the simplifying assumption is made that all types of substitutions are equally likely. This is not necessarily the case in practice, since different techniques for synthesis and sequencing have different error

[5]For example, with an `output_size` of eight and an `input_size` of four, the encoded string would be twice as long as the original string since every four bits will have been encoded into an eight-bit symbol

profiles. This simplified error model is sufficient for assessing the error-correcting capabilities of different encodings produced by the encoding generation scheme.

Injecting errors randomly with a certain error rate

This is the primary method of error injection used throughout this work. An error rate is given, and at each nucleotide, a random floating point number between 0.0 and 1.0 is generated and compared to the rate. If the random number is less than the rate, the nucleotide is randomly flipped to one of the other three nucleotides. Otherwise, the nucleotide is kept the same. This means that no errors are necessarily guaranteed to be injected, however, over a large number of nucleotides, the error rate will tend towards the given rate. The code used to inject random errors can be found in appendix A.3.1.

Injecting an exact number of errors at random locations

An alternative method of error injection experimented with was to inject a precise number of errors at random locations. The advantage of this method is the certainty that any given string has experienced an exact error rate. Initially, this was the favoured approach since many of the core algorithms, such as Viterbi and FSM generation, were written in Python, and therefore experiments took longer, and fewer iterations could be run. This smaller sample size of data points meant variability due to randomness was a big concern, so this error injection method was an attempt to reduce random variability. The code used to inject exact errors can be found in appendix A.3.2. Once the core algorithms had been rewritten in Rust, many more iterations became possible, and so variability due to randomness became less of a concern.

The disadvantage of injecting is precisely that iterations of an experiment are more similar to each other - if a DNA sequence is 200 nucleotides long, and errors are injected at an exact rate of 2%, there will always be four errors in every subsequent iteration. However, if the error rate of 2% is probabilistic, sometimes there may be two, three, five, or six errors. This means that a larger range of errors has actually been tested. This was a downside when the sample size was small since one configuration could, by chance, receive a higher error rate than another, however, with a large sample size, since the average error rate should always be very close to the specified figure, this is actually an advantage. For this reason, once the core algorithms had been rewritten in Rust, injecting errors randomly became the preferred method.

Injecting errors in bursts

An interesting question is how error distributions affect the error correction capacity of the encoding scheme. To test this, another form of error injection was developed, to inject errors in bursts. Unlike the other very simple error injection algorithms, this injection method is somewhat more novel and interesting, so it is included in algorithm 1. The Python implementation can be seen in appendix A.3.3.

Algorithm 1 Burst Error Injection

Input: message: string, rate: float, min_burst: int, max_burst: int
Output: result: string
Require: min_burst $\geq$ max_burst

```
result ← message
num_errors ← ROUND(message.len × rate)
while num_errors > 0 do
    burst ← RANDOMINRANGE(min_burst, max_burst)
    position ← RANDOMINRANGE(message.len - burst)
    error ← ""
    for i ← 0, burst do
        error ← error + RANDOMBASE(exclude=message[position + i])
    end for
    result ← result[:position] + error + result[position + burst:]
    num_errors ← num_errors - burst
end while
```

This algorithm injects error bursts of a length between `min_burst` and `max_burst`. The errors are probabilistic in both location and number. The algorithm works by estimating the number of errors which should be injected based on the length of the string and the given error rate. It then repeatedly creates burst errors of a random length between the `min_burst` and `max_burst` and injects these bursts into random positions throughout the sequence, until it has injected at least the estimated number of errors. The number of errors injected is probabilistic since a burst of length `max_burst` can be injected even if only one more error is needed to reach the estimated error number. It is also possible to get fewer than the estimated error number since there is nothing preventing errors from being repeatedly injected at the same location. The intention behind this probabilistic structure is to make the error profile more varied and realistic.

Injecting insertions and deletions

Tests were also carried out to ascertain if the encoding generation scheme had any capacity for correcting insertion or deletion errors. Insertions and deletions were injected in the same probabilistic way as substitutions, with one nucleotide being inserted or deleted randomly with a given chance. The code used to inject insertions and deletions can be found in appendix A.3.4.

4.3 Performance Problems and Solution

The encoding scheme was initially implemented entirely in Python. This led to an extremely short development time and worked to establish a proof of concept. However, performance limitations quickly led to a few issues. Firstly, performing experiments with long sequence lengths or large symbol sizes was extremely slow. Secondly, performing large numbers of experiment iterations took a very long time (hours to days), which limited the number of data points that could be collected for any given configuration. Once it became clear that the efficiency of this implementation would limit the number and quality of experiments that could be carried out, the decision was made to profile and optimise the code. The approach used was to locate bottlenecks and rewrite them in Rust, a much faster language.

4.3.1 Locating the bottleneck

To avoid wastefully optimising irrelevant code paths, the first step was to locate bottlenecks through profiling. Profiling was carried out using the Python `time` module. Every part of the encoding process was timed with different sequence lengths, including the generation of the finite-state machine, convolutional coding, DNA-to-bit mapping, error injection and Viterbi decoding. It became clear that the bottleneck by far was the Viterbi decoding. The second most time-consuming section was FSM generation. All other sections took less than a tenth of a second, even with sequence lengths in excess of 1000. Figure 4.6 shows the time taken by the Viterbi decoding and FSM generation as a function of sequence length symbol size, respectively. Interestingly, the time taken by Viterbi decoding seems to have a non-linear relationship with sequence length, which is surprising since Viterbi's time complexity is linear with regard to the sequence length. A possible explanation for this is that the Python Viterbi implementation does a large amount of string manipulation, and since strings in Python are immutable, this results in a lot of copying. And since Viterbi follows many possible sequence "paths" to figure out which one is the most likely, the longer the sequence, the more different paths there are, and the more string manipulation needs to be carried out on longer and longer strings. This could potentially explain the results in figure 4.6. For short sequence lengths, both FSM generation and Viterbi decoding form a major portion of the time taken, however, as the sequence lengths increase, Viterbi begins to take up almost all the execution time.

To remedy these bottlenecks, the decision was made to rewrite Viterbi decoding and FSM generation as Rust libraries to be called from Python. Since it is so closely linked to the decoding operation, convolutional coding was also implemented in Rust, despite not being a bottleneck. All other components were kept in Python.

4.3.2 Motivation for choosing Rust

The candidates for the faster implementation of FSM generation and Viterbi decoding initially included C, C++ and Rust. The performance of these languages appears to be similar, and so the decision to use Rust over C or C++ was not to do with speed. The decision was made to not use C on the basis that it lacks generic or template data structures, which are useful abstractions for both FSM generation and Viterbi. Rust was favoured over C++ due to it's better memory safety, type system and more helpful compiler errors. Another advantage of Rust when building Python libraries is a very good ecosystem for creating Python packages (Maturin) and a simple set of Python function bindings (Py03).

4.3.3 Rust implementation and project structure

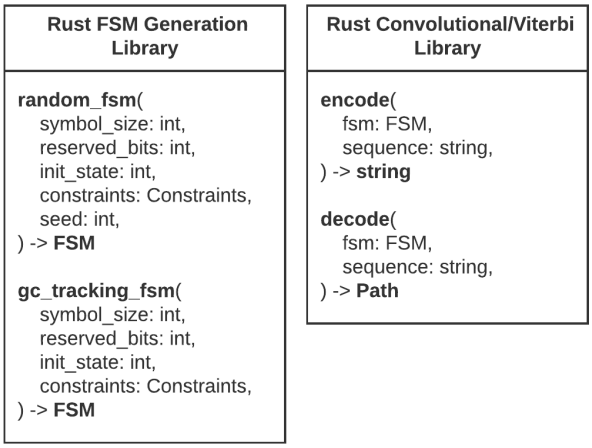
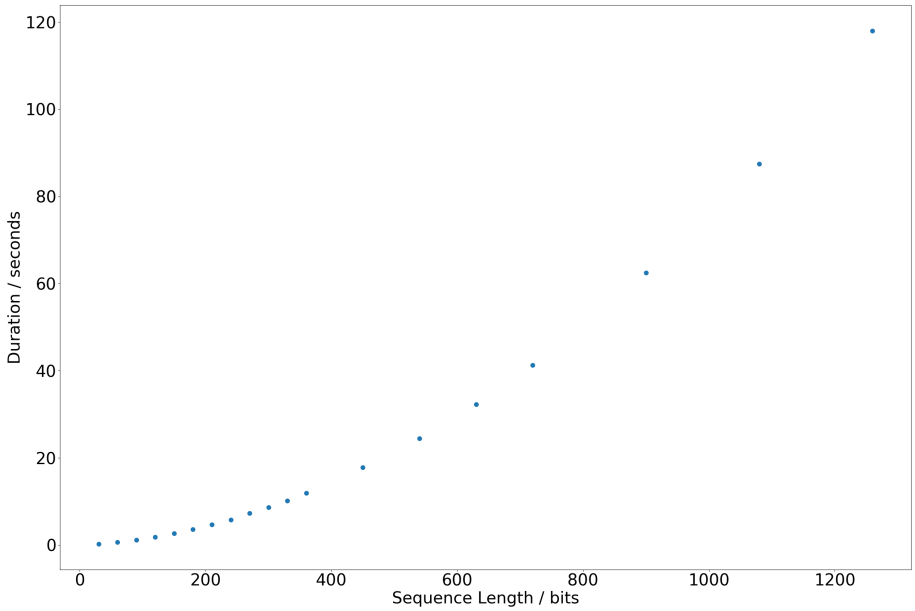


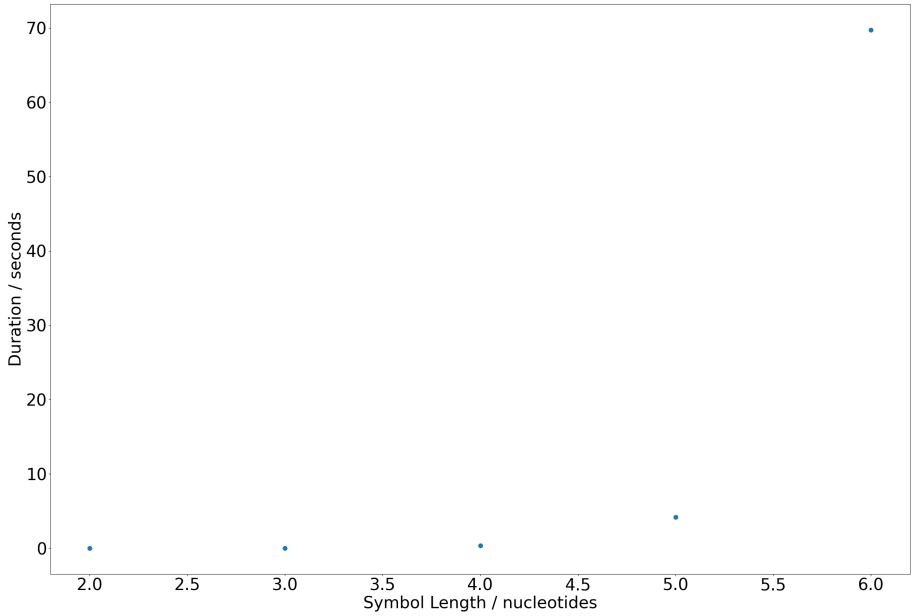
Figure 4.7: API diagrams for the two Rust packages.

In order to keep the project structure clean, it was decided to split the FSM generation and convolutional coding/Viterbi decoding into two separate Rust packages. This approach had a couple of key advantages. Firstly, it forced some amount of decoupling between the two sections, which makes both maintenance and testing easier. Secondly, it meant that the convolutional coding / Viterbi decoding could be totally separate from the DNA-focused part of the code. This meant that the convolutional/Viterbi module could be used for encoding with standard convolutional codes, which again is advantageous for testing, but was also useful during the evaluation portion of this work, in which the error correction capability of the encoding scheme is compared to that of a standard convolutional code.

Both modules have very simple APIs, making them easy to use. Figure 4.7 provides API diagrams for both



(a) Viterbi Decoding



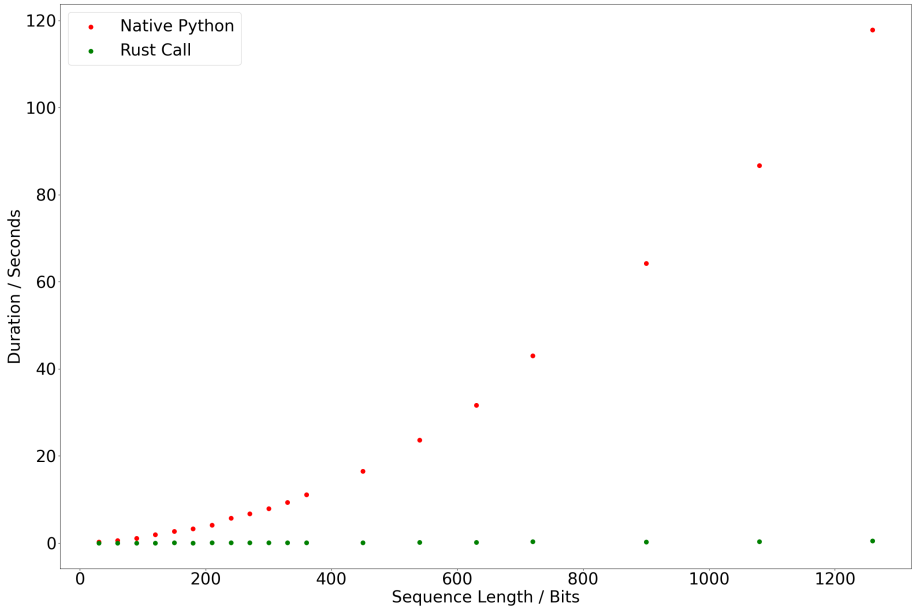
(b) Finite-state Machine Generation

Figure 4.6: Results of profiling the Python implementation

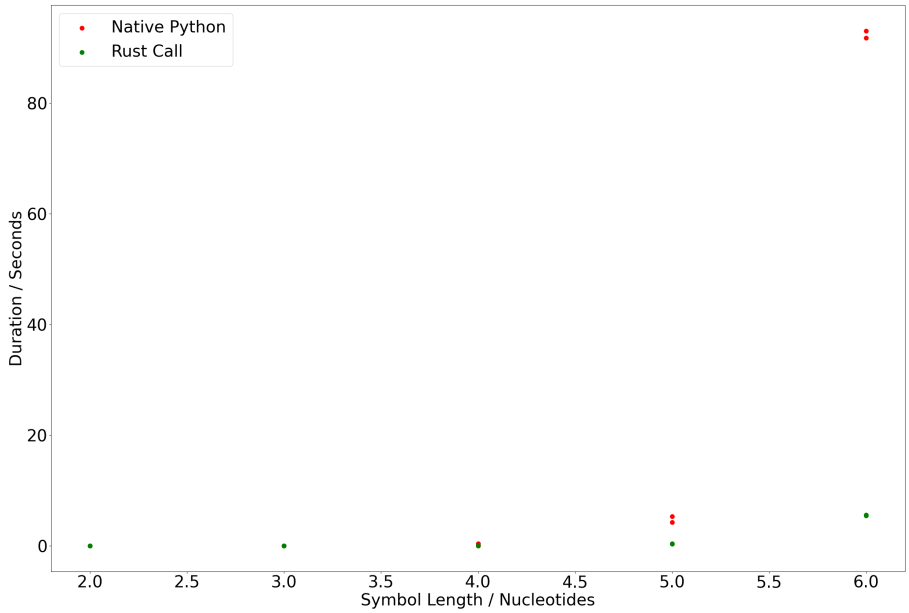
packages. The FSM generation package has functions for generating FSMs using the random choice and GC-tracking mechanisms, both of which return `FSM` data structures. The convolutional coding/Viterbi decoding module has functions for encoding and decoding sequences using a given FSM. The `FSM` data structure is the only connection between the two libraries, limiting the surface area of coupling between them.

4.3.4 Performance Comparison

After completing the Rust libraries, profiling was once again carried out to gauge performance improvements. Figure 4.8 plots the results. These results show an improvement in FSM generation of only a little bit for small symbol sizes, but a difference of over two orders of magnitude for longer symbol sizes. Viterbi decoding has been seriously sped up, by at least two orders of magnitude for longer sequence lengths. Another interesting observation is that the relationship between sequence length and Viterbi decoding duration looks much closer to linear, perhaps due to the more efficient handling of string operations in Rust. These differences are extremely significant, and enable the running of experiments with larger symbol sizes and longer sequences for many more iterations. This performance increase, therefore, massively improved the quality of analysis which could be performed on the encoding scheme. More profiling results can be found in appendix B.



(a) Viterbi Decoding



(b) Finite-state Machine Generation

Figure 4.8: Comparison between Rust and Python profiles

Chapter 5

Evaluating the encoding as an ECC

5.1 Experimental Methodology

In order to access the characteristics of the encoding mechanism, it is necessary to design experiments which measure the output error rate (i.e. the error rate after decoding) against the input error rate (i.e. the rate at which errors were injected into the encoded sequence). To this end, the following experimental methods were used to measure the effect of the encoding on errors.

5.1.1 Experiment Design

In testing the efficacy of the encodings generated with the encoding scheme using different parameters, the following properties are desirable in an experiment design.

- The experiments should be reproducible.
- Each experiment should sweep through a range of error rates.
- Various different hyperparameter configurations should be experimented with in an automated fashion.

A single run

In order to attain an experiment design with the above properties, firstly, the basic unit of all the experiments, the "single run", is defined. This is a single run of the experiment with a given error rate and set of parameters. On all of the following plots, every data point is the result of one single run. Each "single run" is defined by two sets of parameters; firstly, the hyperparameters to be used in the construction of an encoding, and secondly, the parameters defining the design of the experiment.

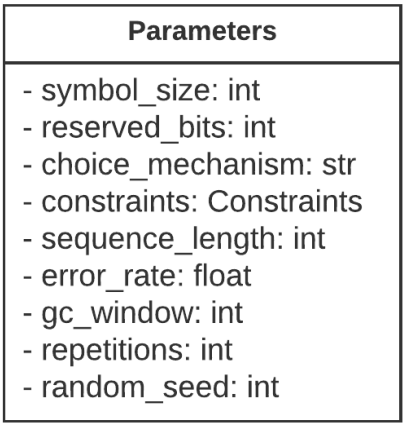


Figure 5.1: A data class containing all the parameters for a single run of an experiment.

Both sets of parameters are captured in a single `Parameters` object, which is used to carry out every single run of an experiment. The encoding hyperparameters are `symbol_size`, `reserved_bits` (the number of reserved bits, not their content), `choice_mechanism` and `constraints`. The remaining attributes of the `Parameters` object are used to define the experimental run itself. For example, each run is seeded with a random seed to ensure that it is reproducible.

A single run has the following structure. Firstly, the `random_seed` is used to seed the experiment. Secondly, the encoding parameters are used to generate an FSM as described in 4.2.3. Next, for each of the given number of `repetitions`, a random bit string of `sequence_length` is generated. Each of these random bit strings is encoded using the generated FSM and converted to DNA using the DNA mapping, as described in section 4.2.1. This resulted in the DNA sequences which would, if this were an actual DNA storage system, go on to be synthesised and then stored. In order to gauge the constraint compliance of the encoding, the GC content and maximum GC variance of the generated DNA sequences are calculated at this point. To simulate the substitution errors which would arise over the course of synthesis, storage, amplification and sequencing, errors are then injected into the sequences at the given `error_rate`. For each of the generated DNA sequences, the input error rate (the rate of errors actually injected) is calculated, which may be slightly different to the given `error_rate` due to the randomised nature of the error injection. After this, each sequence is mapped back into a bit string and then decoded by the Viterbi algorithm using the generated FSM. For each of the decoded bit strings, an output error rate is calculated by comparing them with the original bit string from which they were produced. In order to get a single data point, each of these output error rates is averaged. The same is true for the input error rates. This single run then returns the average values for the input error rate, the output error rate, GC content and maximum GC variance.

The reasons for using several repetitions to produce a single data point are as follows. In this setup, each data point represents the performance of a generated encoding on several strings, rather than representing how a single encoding performs on a particular string. This design should also slightly decrease the variance of the data points, in theory making for a clearer trend. Furthermore, it allows the testing of more encode-decode cycles with fewer FSM generation operations, so a larger number of random strings can be experimented with in a shorter span of time.

The `gc_window` is needed in order to calculate the maximum GC variance of the encoded DNA sequences. This attribute represents the window size used in determining the maximum GC variance - for example, if a company decides that the difference in GC content between any 50 nucleotide stretch of DNA should be no more than 50% (as is the case with IDT[1]), a window size of 50nt should be used to calculate the maximum GC variance, to see if the sequence is below this threshold.

Varying the error rate

For any given set of hyperparameters, it is not sufficient to simply test a single error rate; the aim is to see how the output error rate changes as the rate of errors injected is varied. To this end, Numpy is used to define an error range of interest.

```
error_range = np.linspace(0.0005, 0.02, num=40)
```

This is the default error range used in the majority of the experiments, starting from an error chance of 0.05% and going up to 2% (inclusive) in increments of 0.05%[2]. Clearly, there are 40 error rates in this range, so to do a full sweep of this range, 40 "single runs" are needed, each time with a different value for the `error_rate` parameter. This will result in 40 data points.

Experiment iterations

It is not sufficient, of course, to only have a single data point with a single random seed at each error rate. Therefore, the above operations are repeated for a certain number of iterations. Unlike the `repetitions` in the `Parameters` object, which increases the number of random bit strings used to produce a single data point, each experiment iteration adds more data points to the result. Therefore, if the number of iterations is set to ten, and the default error range is used, the result will contain 400 data points. In order to guarantee that each iteration is different from every other iteration, and to ensure reproducibility, each iteration is seeded with a different random seed. The successive completion of all of these iterations is referred to from here on as an "experiment".

Varying the configurations

One of the desirable experiment properties stated above was that various different hyperparameter configurations should be experimented with in an automated fashion (meaning without having to repeatedly define different configurations). This is achieved by defining the discrete ranges of different hyperparameters which are of interest. For example, one may be interested in values of reserved bits in the range [3, 4, 5], symbol sizes in the range [4, 5] and choice mechanisms in the range ["random", "gc-tracking"]. Once each of these ranges is defined (these definitions must be given manually), an experiment is created for each possible combination of these hyperparameters. For the examples given above, in which there are three values for the number of reserved bits, two for the symbol length, and two different choice mechanisms, 12 different experiments would be created and subsequently run, producing twelve different sets of results.

Random seeds

To guarantee the reproducibility of the results, a list of random seeds is utilised. Each of these seeds was generated by an online random number generator. Each experiment iteration to be run uses the next random seed from the list. This means that each iteration of each experiment is seeded differently from the others (provided that the list is longer than the number of iterations), and that each iteration is reproducible independently from all the others. The random seeds used in each "single run" are produced from the random seed of the wider iteration. All the random seeds used (for an iteration and each single run) are printed as part of the experiment output and subsequently recorded in the results file, such that one could easily produce an identical iteration or single run by using the recorded seed.

Design summary and pseudocode

The experiment design can be summarised as a series of loops; for each configuration, one experiment is run. In each experiment, a given number of iterations is run. Within each iteration, a discrete range of error rates is tested. For each error rate, a single run is carried out, producing a single data point. To produce each data point, a single run generates and tests a number of random bit strings.

To more clearly demonstrate the experiment design, pseudocode is provided in algorithm 2.

[1] Integrated DNA Technologies

[2] This error range is chosen based on a review of the literature concerning error rates in nanopore and next-generation sequencing, such as [44] and [45].

Algorithm 2 Experiment Design

```
iterations ← 10                                              ▷ Set number of iterations
seeds ← [...]                                                ▷ Randomly generated list of random seeds
configs ← [...]                                              ▷ Configurations, generated from hyperparameter ranges
error_range ← [0.0005, 0.001, 0.0015, ..., 0.02]
for c in configs do
    input_errors ← []
    output_errors ← []
    gc_contents ← []
    gc_vars ← []
    for i ← 1, iterations do
        seed ← next(seeds)
        for rate in error_range do
            fsm ← generate_fsm(c.params)                     ▷ Generate FSM using config hyperparameters
            avg_in_err ← 0.0
            avg_out_err ← 0.0
            avg_gc_content ← 0.0
            avg_gc_var ← 0.0
            for i ← 1, c.repetitions do                      ▷ Number of repetitions defined in config
                seq ← RANDOM_STRING(c.sequence_length)
                enc ← ENCODE(seq, fsm)
                dna ← BITS_TO_DNA(enc)
                err ← INJECT_ERRORS(dna, rate)
                dec ← DECODE(err, fsm)                       ▷ Decode the received sequence
                est ← DNA_TO_BITS(dec)                       ▷ Maximum likelihood estimate for sequence

                avg_in_err += HAMMING_DISTANCE(dna, err) / c.sequence_length   ▷ Input error rate
                avg_out_err += HAMMING_DISTANCE(seq, est) / LEN(dna)           ▷ Output error rate
                avg_gc_content += GC_CONTENT(dna)
                avg_gc_var += MAX_GC_VAR(dna)
            end for
            avg_in_err ← avg_in_err / c.repetitions
            avg_out_err ← avg_out_err / c.repetitions
            avg_gc_content ← avg_gc_content / c.repetitions
            avg_gc_var ← avg_gc_var / c.repetitions

            input_errors += avg_in_err                       ▷ Append to list of results
            output_errors += avg_out_err
            gc_contents += avg_gc_content
            gc_vars += avg_gc_var
        end for
    end for
    PRINT_RESULTS(c, input_errors, output_errors, gc_contents, gc_vars)   ▷ Results of this config
end for
```

5.1.2 Measuring Errors

All error measurements in this work are taken in terms of error rates rather than a raw number of errors. This is required for a few reasons. Firstly, to simulate DNA operations, errors are injected into the encoded nucleotide sequences, but the output errors of interest are in terms of bits (the difference between the original and decoded sequences). Therefore, the input errors are symbol errors whereas the output errors are bit errors - meaning comparing the raw number of errors is not a useful thing to do - so the error rates must be compared. Secondly, a change in the values of certain hyperparameters will result in a change in the length of the encoded dna sequence. For example, increasing the number of reserved bits from four to five will result in a greater coding overhead, leading to a longer encoded sequence. Hence, when comparing the error correction ability of two configurations with different numbers of reserved bits, it is necessary to use the error rate rather than the number of injected errors.

5.1.3 Parameters Varied

Throughout the experiments described in this work, as well as varying the error rate, the following parameters have been varied.

- Choice mechanism
- Number of reserved bits, i.e. coding overhead
- The set of constraints
- Symbol length
- The type and distribution of errors

It is worth noting that varying parameters such as symbol length and number of reserved bits sometimes require a corresponding change in the constraints. For example, with a symbol length of four and the number of reserved bits set to five, GC limits of 35% and 65% are satisfiable. If the number of reserved bits is decreased to three, these GC limits can no longer be met and must be changed to 15% and 85%. In a setting where meeting certain constraints is the primary aim, such as in Sella *et al.*'s original paper, it makes sense to start from the constraints and derive the minimum number of reserved bits. However, since this work is focused on error correction, for which reserved bits are required regardless of constraints, the direction of this arrow is reversed, and the following question is considered; "Given that this many reserved bits are needed to meet the error correction requirements, what are the strictest constraints which can be met?".

5.2 Experimental Results

5.2.1 Assessing and minimising the error rate

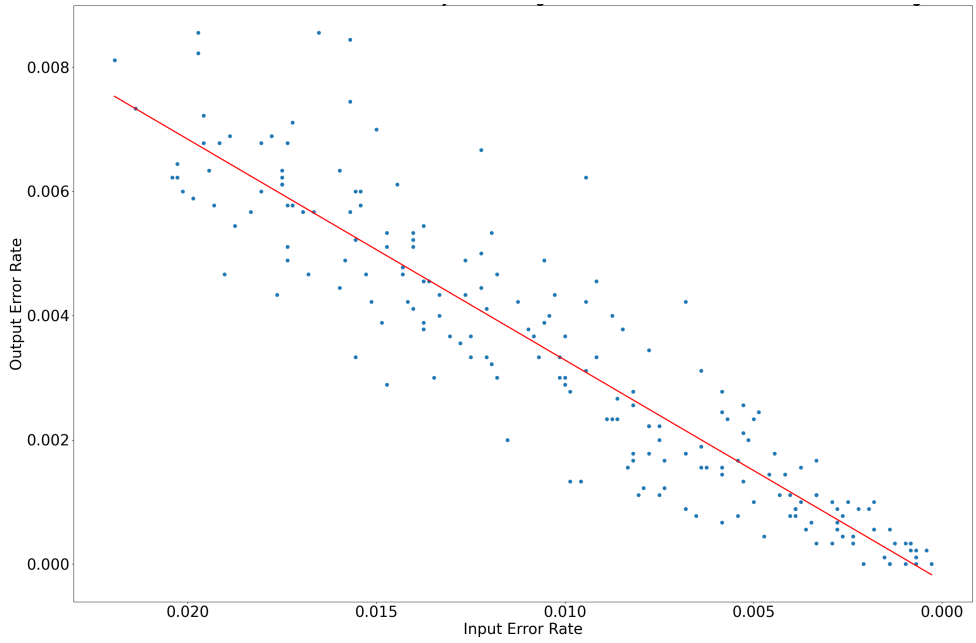


Figure 5.2: Results of default configuration with the random choice mechanism.

To begin assessing the efficacy of the encodings generated by the encoding scheme as ECCs, a "default" configuration is chosen; a symbol length of four, three reserved bits, a maximum homopolymer length of eight and GC

limits of 15% and 85%. This default is chosen as it aligns with some of the constraints presented by various DNA manufacturers. The first experiment carried out measures the effect of this encoding scheme on error rate with both the random and GC-tracking choice mechanisms. The results are shown in figures 5.2 and 5.3.

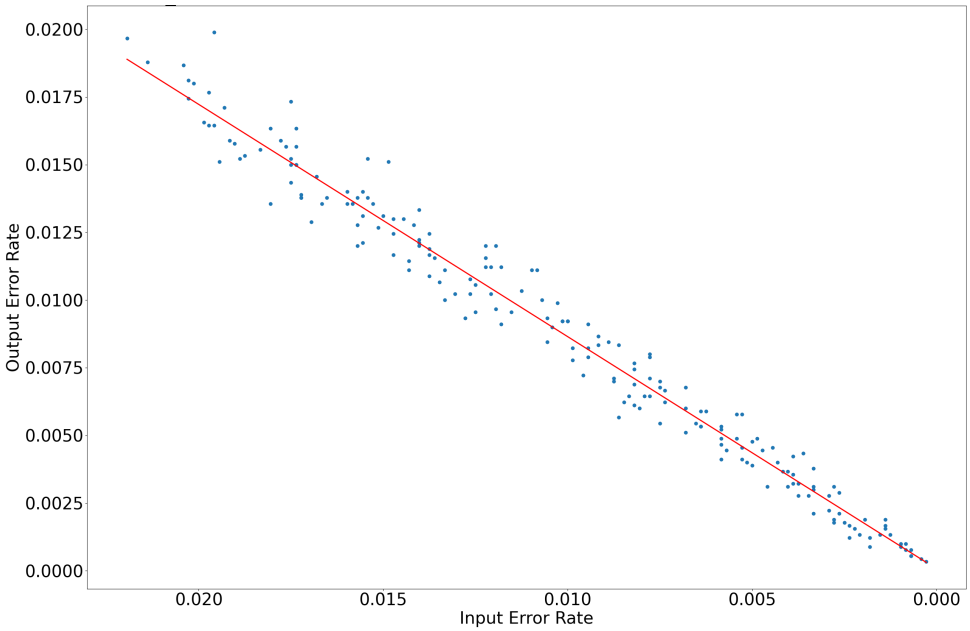


Figure 5.3: Results of default configuration with GC-tracking choice mechanism.

In comparing the two plots above, two observations are very clear. The first is that encodings generated by the encoding scheme clearly do have some efficacy as ECCs. This is made plain by figure 5.2 in which input error rates of 0.02 (2%) and 0.01 (1%) seem to correspond with output error rates of 0.007 (0.7%) and 0.003 (0.3%) respectively. The second observation is that when using the GC tracking choice mechanism, there is little to no error correction occurring. This suggests that there is a trade-off between being able to meet GC variation constraints and being able to correct errors when using this encoding. The ability of the GC-tracking choice mechanism to decrease GC variation is gauged in section 5.2.4. In section 6.1.1, a new choice mechanism is proposed to decrease GC variance whilst maximising error correction.

To more clearly demonstrate the magnitude of the difference between the two choice mechanisms, figure 5.4 shows them on the same axis.

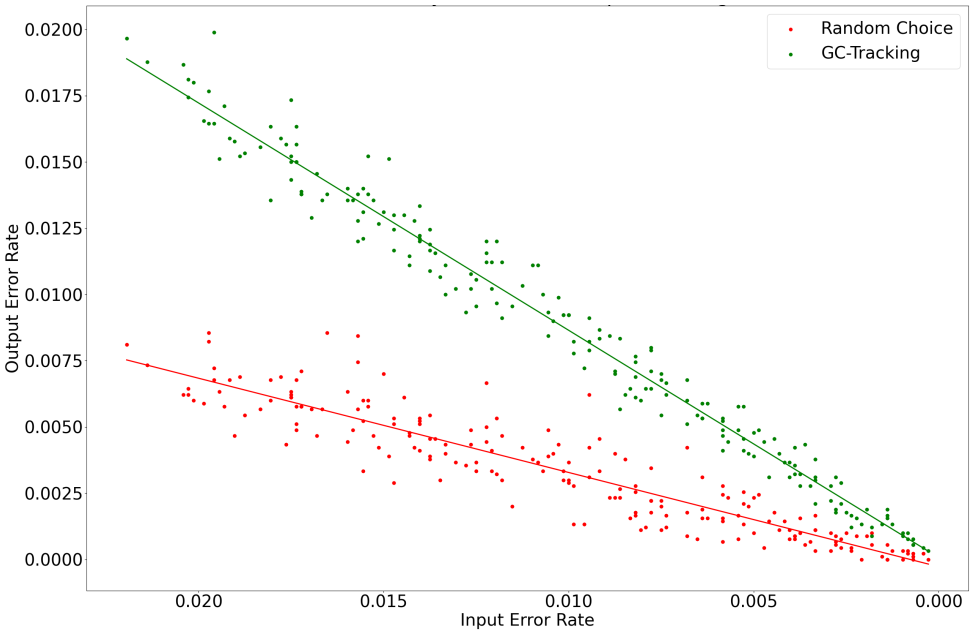


Figure 5.4: Comparing random choice with GC-tracking using the default configuration.

Having established that using the random choice mechanism results in some ability to correct substitution errors, the extent to which error correction can be improved by increasing the coding overhead is tested. Figure 5.5 plots

the error rates of the default configuration with three, four and five reserved bits. In all cases, the symbol length is kept at four, so the increase in the number of reserved bits represents an increase in coding overhead. With each increase in reserved bits, the GC limits are also modified to 25% and 75% followed by 35% and 65% since those are the strictest constraints which can be met by four and five reserved bits, respectively. Note that the y-axis in figure 5.5 is stretched compared to figure 5.4 and that the red points from these graphs represent the same data.

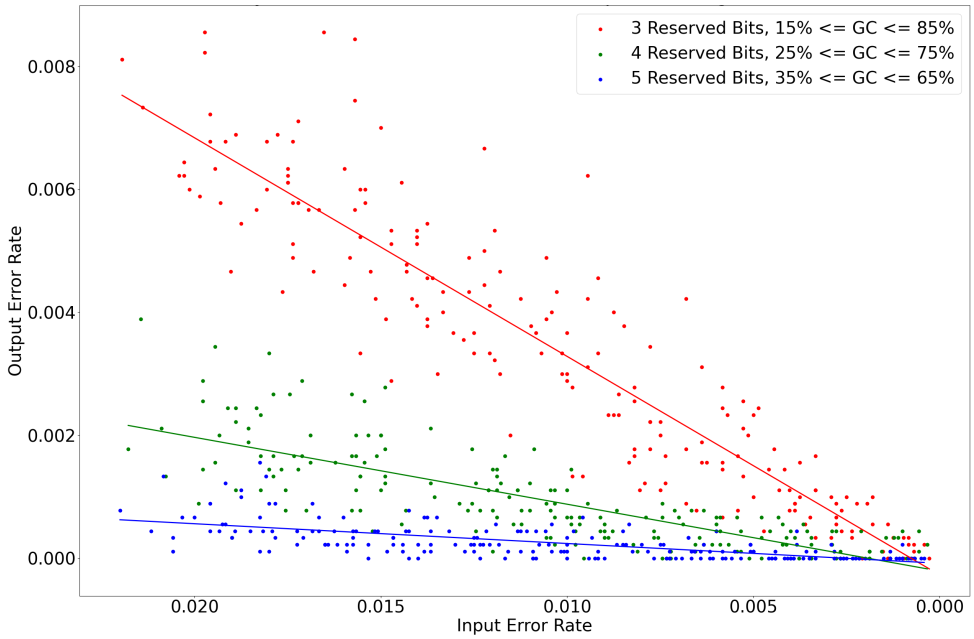


Figure 5.5: Comparing 3, 4 and 5 reserved bits using the random choice mechanism. All other parameters are set to default.

This result demonstrates that there is a large jump in error correction when increasing from three to four reserved bits, and another smaller yet still significant jump when increasing from four to five reserved bits, despite the fact that the constraints are getting stricter at the same time.

Next, an experiment is run to determine whether the number of reserved bits can do anything to improve the poor error correction observed when using the GC-tracking choice mechanism. Figure 5.6 compares the results of varying the reserved bits when using either the random or GC-tracking choice mechanisms. What is observed is that the number of reserved bits makes very little difference to error correction when the GC-tracking mechanism is used; the encodings produced with the choice mechanism simply have no capacity for error correction.

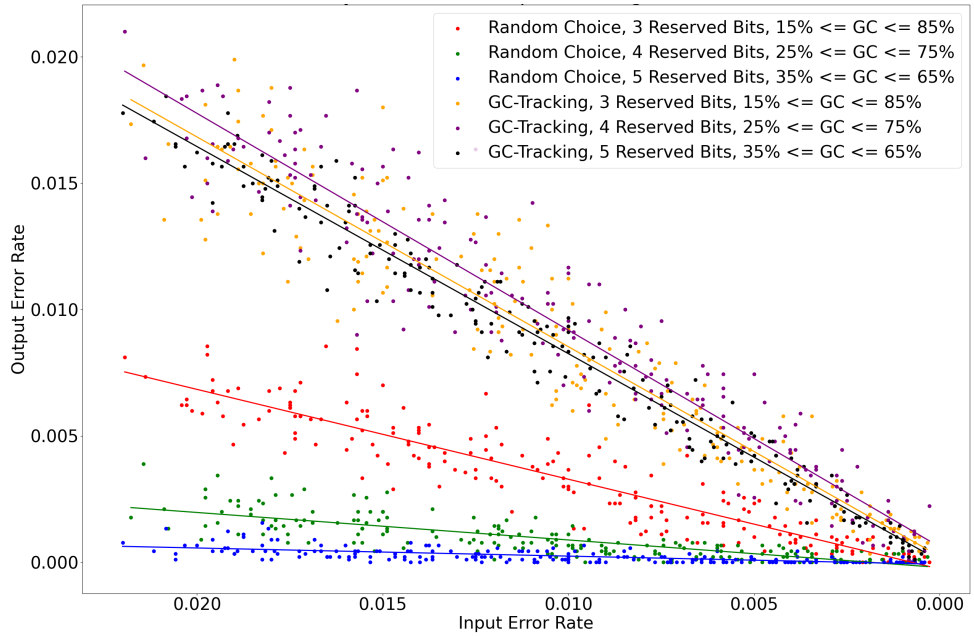


Figure 5.6: Comparing 3, 4 and 5 reserved bits using GC-tracking. All other parameters are set to default.

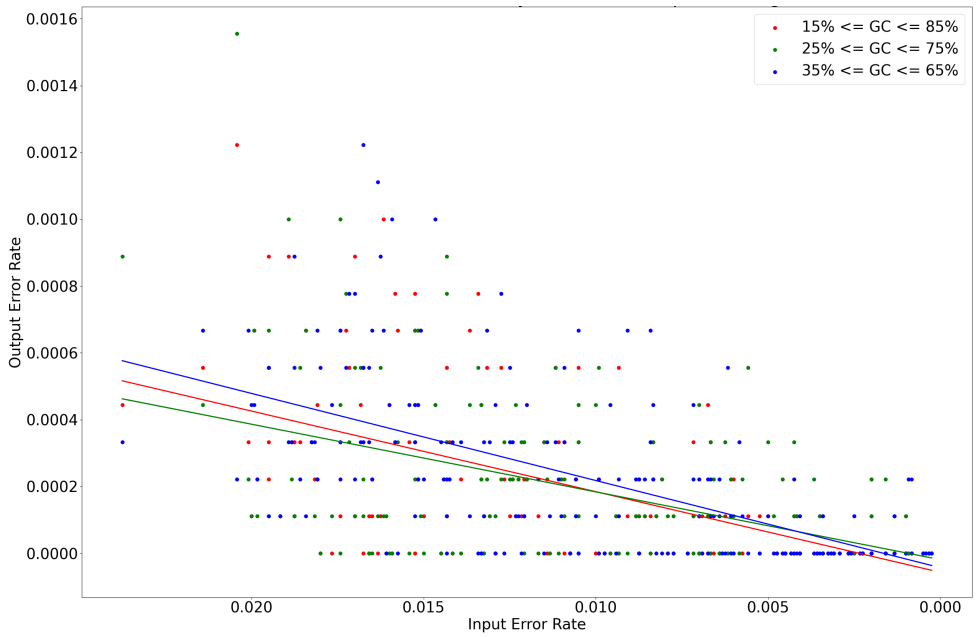


Figure 5.7: Comparing GC limits of 15%-85%, 25%-75% and 35%-65%. There are 5 Reserved bits, and all other parameters are set to default.

The discovery that an increase in reserved bits increases error correction despite the stricter constraints prompts an interesting question - how much of a trade-off is there between meeting constraints and correcting errors? As the strictness of the GC limits was increased, how much error correction is lost out on? To answer this, an experiment is perfomed that only varies the GC limit constraints without varying the number of reserved bits. The number of reserved bits is set to five, and the GC limits are varied between 15%-85%, 25%-75% and 35%-65%. The departure from Sella *et al.*'s method of deriving the number of reserved bits from the given constraints was precisely to enable this type of experiment, in which the constraints and number of reserved bits are decoupled. The results of this experiment can be seen in figure 5.7.

The results in figure 5.7 suggest that this loosening in constraints has a minimal effect on error correction - it is likely that most of the variance between these results is due to chance, since the lowest error values come from GC limits of 25-75%, which is in between the other two ranges. This suggests that there is little to be gained from loosening up the constraints whilst keeping the number of reserved bits the same.

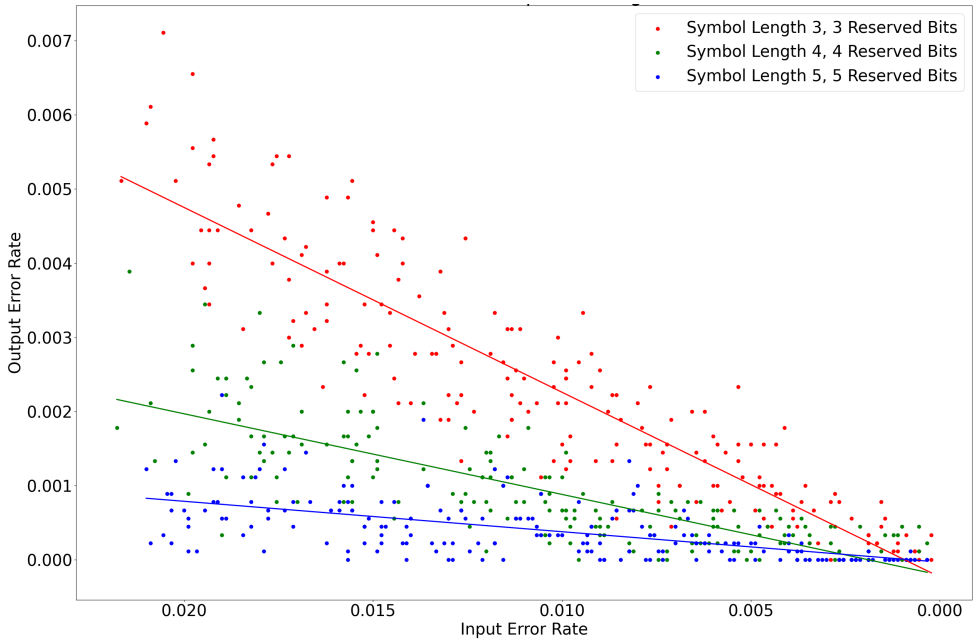


Figure 5.8: Comparing different symbol lengths with the same coding overhead. In all cases, half of the available bits are reserved.

Another important hyperparameter in defining the encoding is symbol length. Unlike the number of reserved

bits or the choice mechanism, the symbol length has a major effect on the performance of the encoding scheme. The Viterbi algorithm used for decoding has a time complexity of $\mathbf{O(T \times S^2)}$ where **T** is the length of the sequence being decoded and **S** is the size of the state space. The number of states is determined by the symbol size, since each symbol FSM is a state in the FSM. Therefore, the relationship between the size of the state space and the symbol length is $\mathbf{S = 4^L = 2^{2*O}}$ where **L** is the symbol length and **O** is the output length (i.e. the symbol length in bits rather than nucleotides). Hence the Viterbi algorithm has a time complexity of $\mathbf{O(T \times 4^{2L}) = O(T \times 8^L)}$ which is exponential with respect to the symbol length. For this reason, it is not practical to use very large symbol sizes. In this work, symbol sizes in the range [3, 5] are experimented with. It should also be noted that symbol length and the number of reserved bits need to be varied in tandem to keep coding overhead the same - as the symbol size increases, the number of reserved bits needs to be increased in lockstep. Figure 5.8 demonstrates the result of changing the symbol size and the number of reserved bits together. Each configuration uses exactly half of the available bits in a symbol as reserved bits, keeping coding overhead constant. This experiment shows that, when coding overhead is kept constant, larger symbol sizes result in better error correction; however, the cost of extra encoding and decoding time must be paid for this improvement.

To demonstrate why reserved bits must be varied with symbol size, figure 5.9 plots the same three symbol sizes, but with the number of reserved bits kept constant. Four reserved bits were used in this experiment. The effect is that as symbol size increases, the proportion of a symbol which is reserved decreases, resulting in worse error correction.

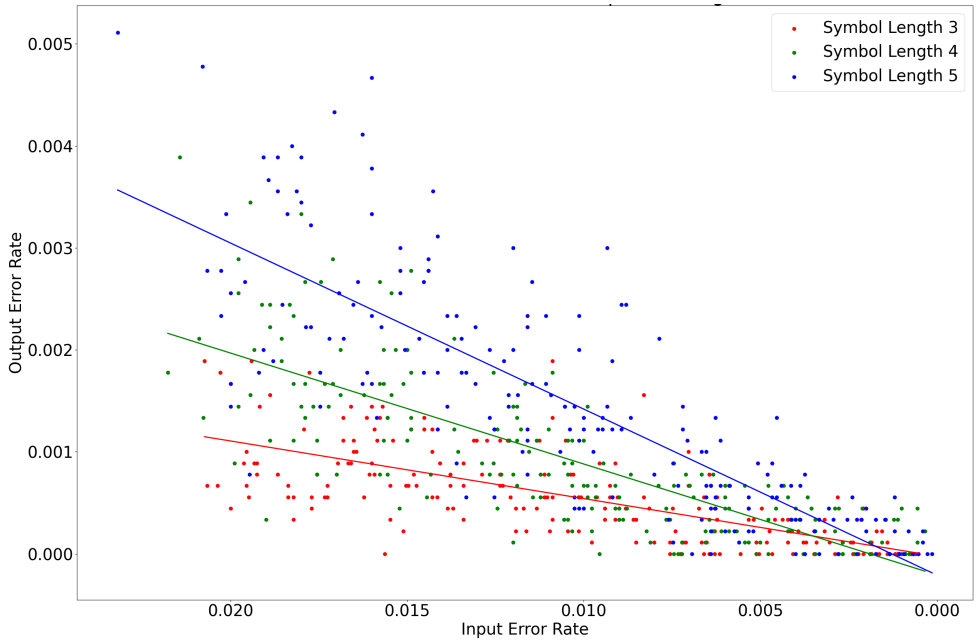


Figure 5.9: Comparing different symbol lengths while keeping the number of reserved bits the same. Here the coding overhead decreases with every increase in symbol length.

The various trade-offs between coding overhead, error correction ability and decoding speed prompt the question of finding an optimal balance between these factors. The experiments carried out in this work suggest that there is no optimal configuration, but instead, a set of guidelines can be offered depending on the speed requirements and error characteristics of a given application.

Figure 5.10 demonstrates a number of configurations which could be considered to strike a good balance. The best performing of these is with a symbol length of four and five reserved bits. Since four nucleotides are equivalent to eight bits, $5/8 = 62.5\%$ of the bits are reserved, so only 37.5% of the bits carry data. In return for this high overhead, this configuration delivers both speed in encoding and decoding and good error correction performance. Setting the symbol length and number of reserved bits to five delivers error correction that is almost as good as a symbol length of four, but at a lower coding overhead; $5/10 = 50\%$, allowing for lower synthesis/sequencing costs and higher information density. However, this comes at the price of encoding/decoding speed due to the larger symbol size. If speed and coding overhead are both key, a symbol length of four could be paired with four reserved bits. This configuration results in a coding overhead of $4/8 = 50\%$ and high encoding/decoding speeds at the cost of worse error correction. Finally, if coding overhead must be kept very low, and both error correction and speed are secondary, a good configuration could be to set the symbol length to five and the number of reserved bits to four. This will result in an overhead of $4/10 = 40\%$. All of these configurations have better error correction performance than the initial default configuration (symbol length 4, 3 reserved bits), but this default configuration does have the advantage of being both fast and having lower overhead.

Figure 5.11 represents this trade-off diagrammatically and offers suggestions depending on the aims of a particular encoding. **L** and **R** represent the symbol length and number of reserved bits, respectively.

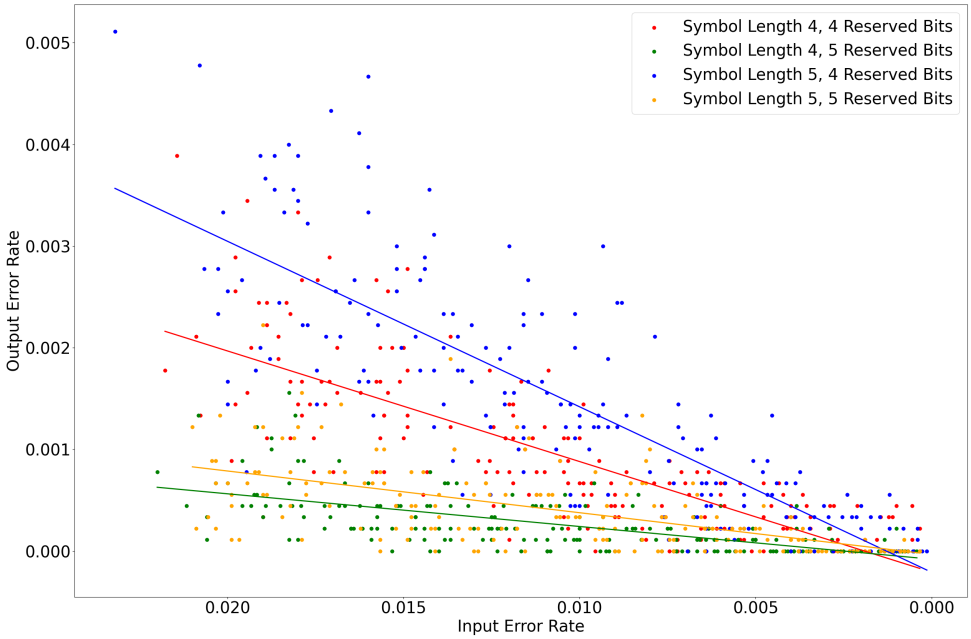


Figure 5.10: A comparison of four “good” configurations.

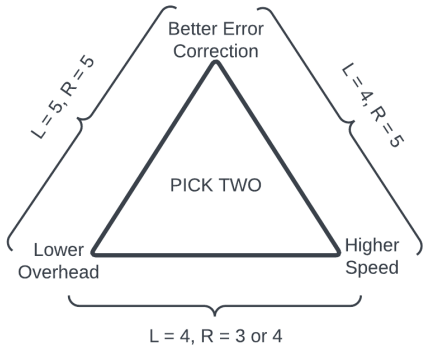


Figure 5.11: The trade-off between speed, error correction and overhead.

5.2.2 Burst Errors

As the encoding scheme is based on convolutional codes, error correction is “local” in the sense that all the redundant information is stored in the same location - no redundant information is stored in a separate or non-adjacent part of the DNA sequence. This means that the code is very vulnerable to burst errors, i.e. a number of consecutive errors. To assess the effect that burst errors have on error correction, the burst error injecting algorithm described in section 4.2.6 is used. Figure 5.12 demonstrates that error correction capacity degrades rapidly as the length of error bursts increases. This plot shows that the output error rate gets much worse as error bursts get longer, even while the input error rate is not changing. This experiment used a symbol length of four; notice that as the burst length increases beyond the symbol length, there is essentially no longer any error correction. This is expected since the more of a symbol is taken up by erroneous bits, the more likely Viterbi becomes to mispredict the input that produced it. This implies that longer symbol lengths will improve the handling of burst errors. Unfortunately, the use of much longer symbol lengths results in unacceptably low decoding times.

5.2.3 Insertion and deletion errors

The encoding generation scheme is not able to correct insertion and deletion errors at all. The reason for this is that it relies on the standard Viterbi algorithm, which does not handle insertions or deletions. If a nucleotide is inserted or deleted, the symbol boundaries are shifted, meaning that all symbols after an insertion or deletion will be affected. For dealing with these types of errors, there are two possible solutions. The first is to discard all oligonucleotides which are not the correct length. This is reasonable due to the massively parallel nature of DNA operations - there are likely to be many copies of the same sequence, so it is possible to discard oligos and still sequence all the data. This approach will not handle cases when a deletion has occurred followed by a substitution at another location since these two errors combined will not change the length of the DNA strand. The second approach (which is not mutually exclusive to the first) is to include another ECC code which can handle deletions. For this reason, future work includes the integration of this encoding scheme with other ECCs.

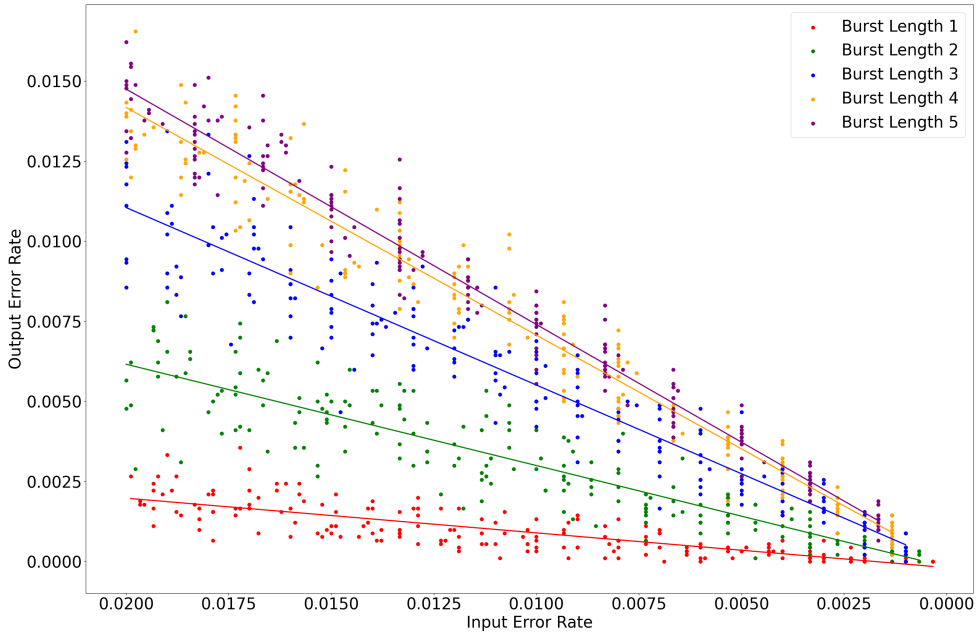


Figure 5.12: The effect of burst errors on error correction.

5.2.4 Complying with constraints

The encoding mechanism guarantees compliance with the constraints used to construct the FSM. These are those given to the FSM generation algorithm in the `Constraints` object. Therefore, it is guaranteed that no encoded sequence will contain a homopolymer longer than the maximum run length, a reserved subsequence and that the global GC content will be within the specified limits (since the GC content of every consecutive pair of symbols is within this limit, the global GC content must also be). However, the `Constraints` object provides no guarantees concerning the GC variation constraint. In this work, it is hypothesised that the GC-tracking choice mechanism can be used to control GC variation, and reliably bring it below a reasonable threshold.

In order to measure compliance with the GC variation constraint, a window size needs to be selected. The window size is simply the length (in nucleotides) of the subsequence which will be used to measure GC content. The maximum GC variation in a DNA sequence is then the difference in GC content between the subsequences with the highest and the lowest GC contents. Since global GC content is limited to be within the range specified in the constraints, the larger the window size used, the lower the GC variance. In this work, window sizes ranging from 10 to 50 are investigated, as values in this range are common among DNA manufacturers. For window sizes longer than 50, the random choice mechanism provides sufficient GC variance control, so GC-tracking is not needed.

Figure 5.13 demonstrates that the GC-tracking choice mechanism is highly effective at reducing GC variation; the largest GC variation between subsequences of ten nucleotides when using GC-tracking, was less than the smallest GC variation observed between subsequences of 50 nucleotides when using the random choice mechanism. This is a highly desirable characteristic to have since many DNA manufacturers stipulate a limit to GC variation.

Figure 5.14 shows the effect that varying the number of reserved bits has on GC variation. An interesting result is that, when using the random choice mechanism, more than half the bits being reserved seems to provide a moderate improvement. The most likely explanation for this is that increasing the number of reserved bits allows for stricter GC content limits, which in turn limits the GC variation to a degree. It is clear, however, that GC-tracking provides superior GC variation control with three reserved bits than random choice does with five bits. A further interesting result is that increasing or decreasing the number of bits does not seem to have an effect on the GC variation when using GC-tracking. This is not particularly surprising as the range of reserved bits (three to five) is not particularly large - if the number of reserved bits was decreased to one or two, greater GC variation could be expected. Therefore, it seems that provided the number of reserved bits is sufficient, adding more reserved bits will not decrease GC variation.

These results suggest that there is a binary trade-off between achieving error correction, for which the random choice mechanism is required, and decreasing GC variation, for which the GC-tracking choice mechanism is needed. However, this work goes on to show that this is not the case in section 6.1.1, in which a new choice mechanism called randomised GC-tracking is proposed.

5.3 Conclusions regarding the encoding scheme

This chapter has demonstrated that the encoding generation scheme described by Sella *et al.* [13] with some minor modifications can be used to correct substitution errors, whilst keeping the constraint compliance benefits of the scheme. Furthermore, the efficacy of the GC-tracking choice mechanism in reducing GC variance has been established. It has also been shown that the use of the GC-tracking choice mechanism negates the error correction

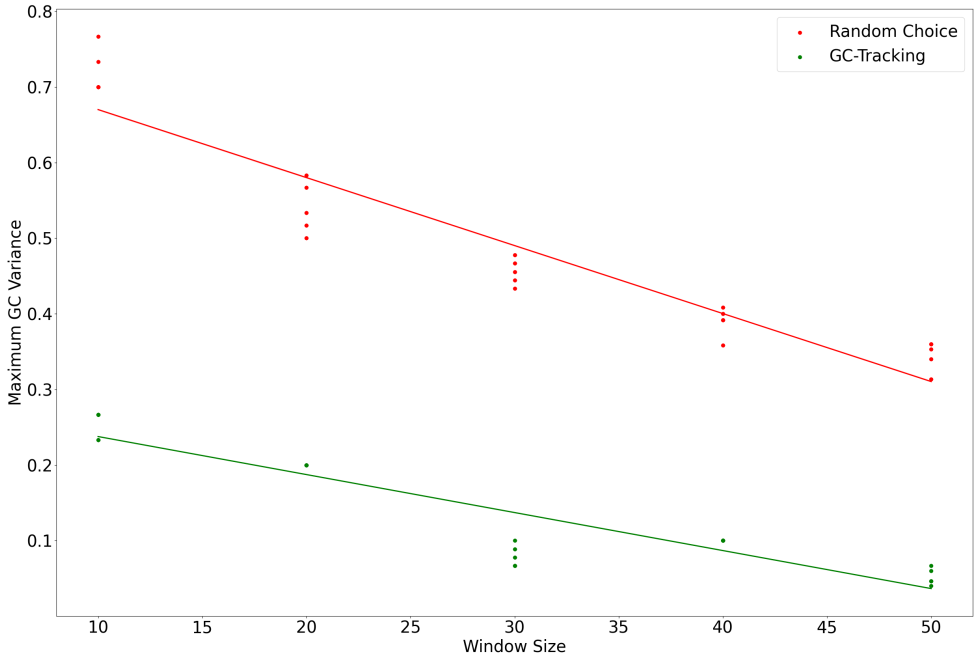


Figure 5.13: Decrease in GC variation due to GC-tracking. Symbol Length four, four reserved bits.

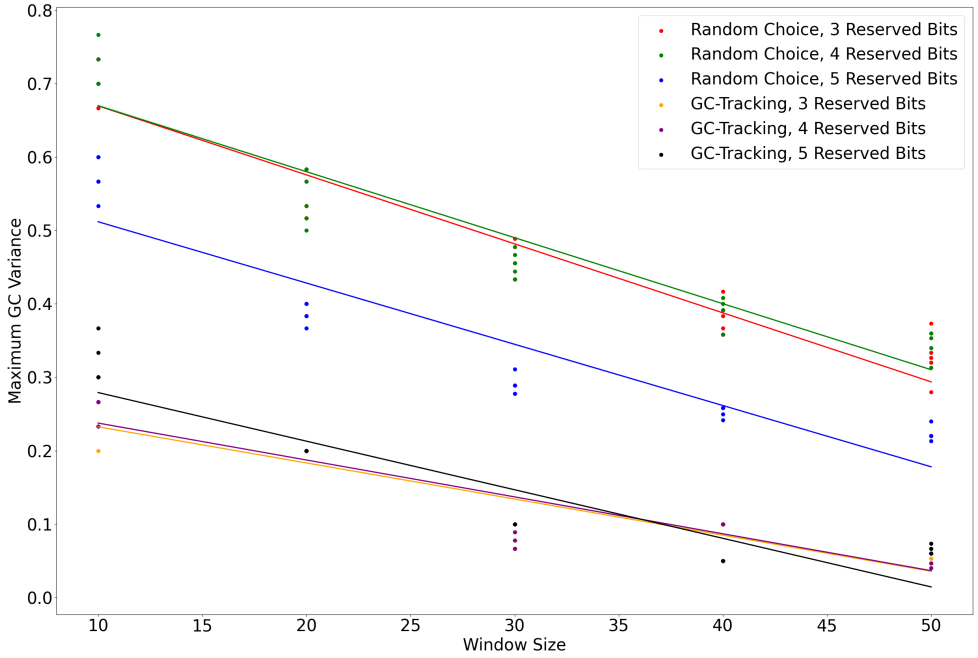


Figure 5.14: The effect of reserved bits and choice mechanism on GC variation. A symbol length of four is used.

properties of the scheme. The limitations of the scheme with regard to insertion, deletion and burst errors have also been demonstrated.

Chapter 6

Extending the Encoding Scheme

In extending the encoding generation scheme described by Sella *et al.*, there are three primary aims:

- To combine the GC variation control provided by the GC-tracking choice mechanism with the error correction performance of the random choice mechanism.
- To improve the error correction characteristics of the generated encodings.
- To partially or entirely mitigate other biochemical constraints such as STRs.

In addressing the first of these aims, an intuitive approach is to try and combine the two choice mechanisms, which is the subject of section 6.1.1. This turns out to work rather well, achieving all the GC variation control of GC-tracking whilst recovering a significant portion of the random choice mechanism's error correction capabilities.

The second aim turns out to be more elusive; a number of different possible techniques have been experimented with, both in terms of alternative choice mechanisms and alterations to the decoding algorithm, with varying levels of success.

With respect to the third aim, a novel mechanism is developed whereby STRs are added to the constraints object mentioned in chapter 4. This mechanism allows for the removal of many STRs but does not guarantee the elimination of STRs entirely. It is described in section 6.3.

6.1 Choice Mechanisms

6.1.1 Randomised GC-tracking - the best of both worlds?

Algorithm 3 Randomised GC-Tracking

Input: state: string, input: string, reserved: list[string], seed: int
Output: chosen: string

```
closest ← []
min_diff ← 1.0
gc_target ← 0.5
for i ← 0, reserved.len do
    candidate ← reserved[i]
    concat ← state + candidate + input
    dna ← BITS_TO_DNA(concat)
    gc_content ← GC_CONTENT(dna)
    diff ← ABS(gc_target - gc_content)
    if diff < min_diff then
        min_dif ← diff
        closest ← [candidate]
    else if diff == min_diff then
        closest ← closest + candidate
    end if
end for
chosen ← RANDOMCHOICE(closest, seed)
```

Randomised GC-tracking is an entirely novel choice mechanism in which, rather than picking the first set of reserved bits that produces the optimal GC content, all possible choices for the reserved bit values which have an optimal GC content are recorded and selected between randomly. This choice mechanism is defined by algorithm 3, and the Rust implementation can be found in appendix A.4.2.

The randomised version of GC-tracking turns out to have far better error-correction ability than standard GC-tracking, whilst not being quite as good as random choice in that regard. However, like GC-tracking and unlike random choice, randomised GC-tracking appears to also be able to seriously limit the GC variation in the encoded

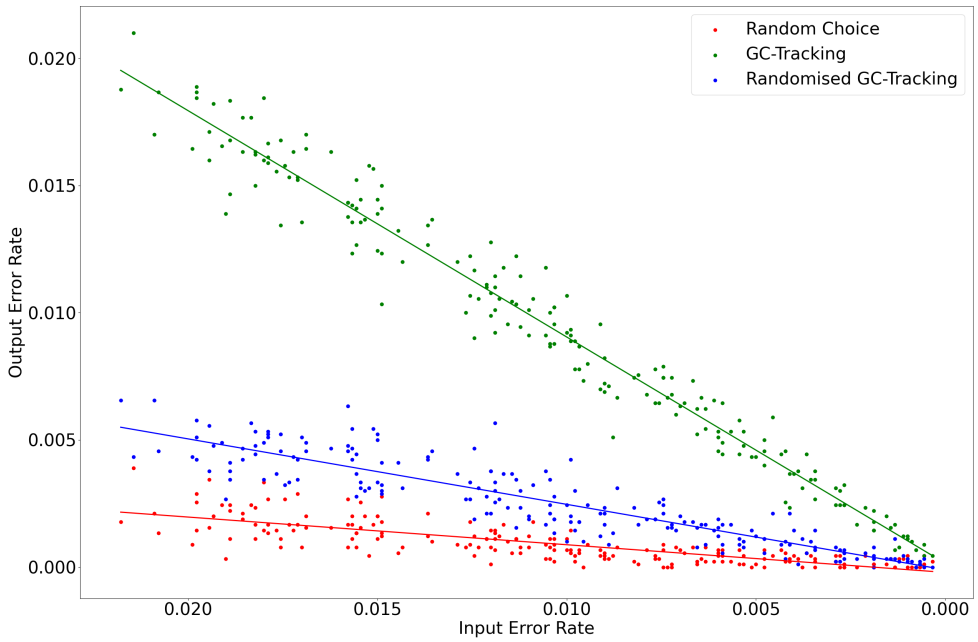


Figure 6.1: Comparison between random choice, GC-tracking and randomised GC-tracking. The symbol size and number of reserved bits are both four.

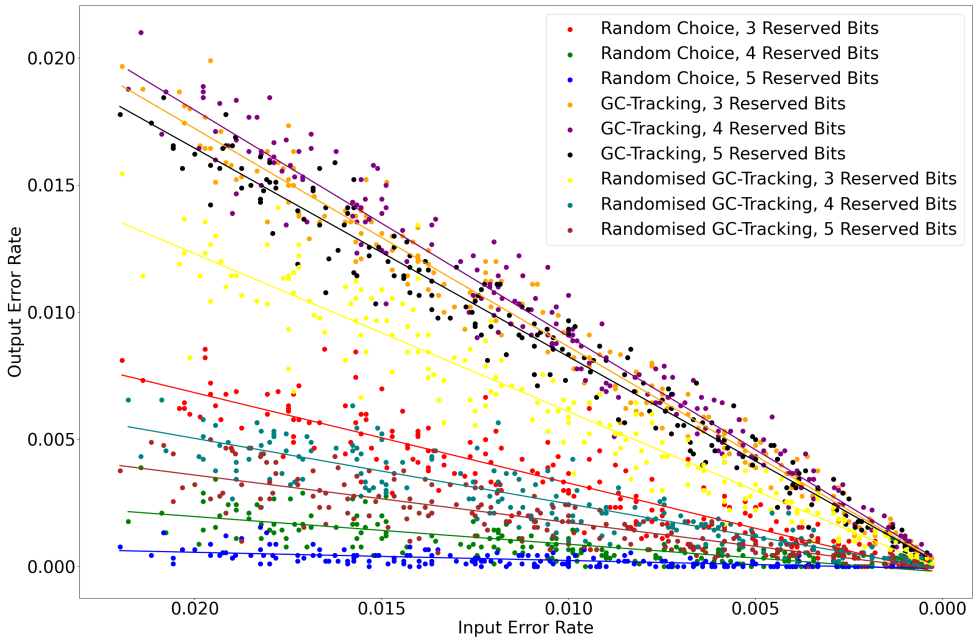


Figure 6.2: Comparison between random choice, GC-tracking and randomised GC-tracking. The symbol size is four.

string. Figure 6.1 plots the error rates which result from using GC-tracking, randomised GC-tracking and random as the choice mechanism. As the number of reserved bits is increased, the error correction performance of encodings generated with both randomised GC-tracking and random choice improves, whereas GC-tracking continues to display little to no error-correcting ability. Figure 6.2 plots the error characteristics of all three choice mechanisms with three to five reserved bits with a symbol length of four. An interesting observation to be made from this figure is that randomised GC-tracking with three or four reserved bits performs better than the random choice mechanism with five reserved bits. It also shows that randomised GC-tracking with three bits performs better than GC-tracking with any number of reserved bits. This plot makes it clear, therefore, that in switching from using the random choice to using randomised GC-tracking, there is definitely a decrease in error correction.

Whilst recovering most of the error correction of the random choice mechanism, randomised GC-tracking also seems to maintain all of the GC variation control of GC-tracking. Figure 6.3 demonstrates that randomised GC-tracking, as expected, has the same GC variation profile as GC-tracking. Thus there is no longer any advantage to using the standard GC-tracking choice mechanism.

In light of this new data, some recommendations can be provided depending on priorities. If the only priority is error correction with minimal coding overhead, and GC variation is not a concern, the random choice mechanism

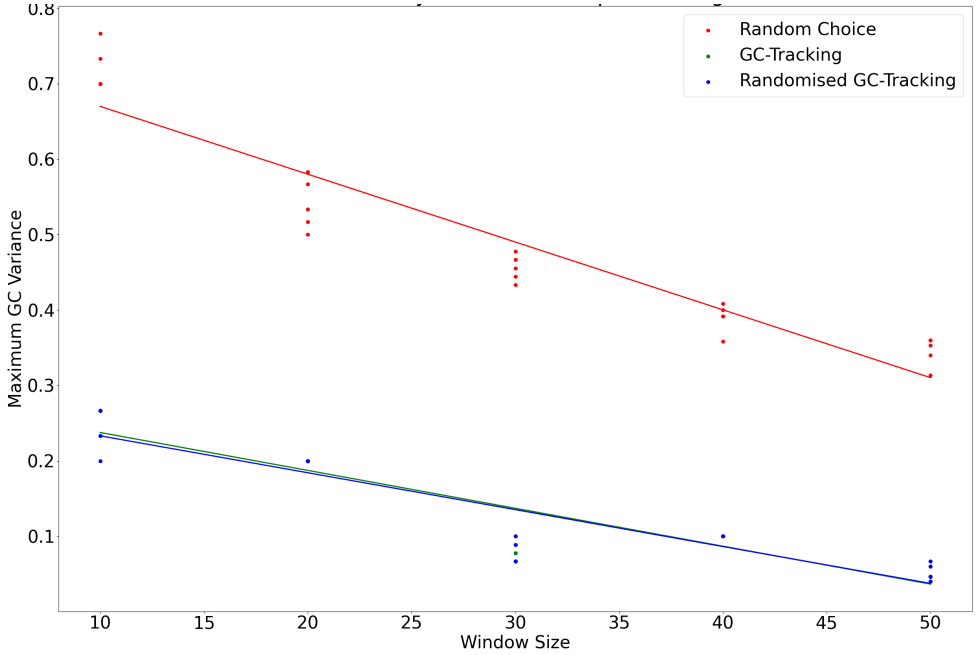


Figure 6.3: Comparison of the GC variation control resulting from the use of random, GC-tracking, and randomised GC-tracking choice mechanisms. The symbol size and number of reserved bits are both four.

should be used, as it provides the best error correction for any given level of coding overhead. However, if GC variation becomes an issue, the randomised GC-tracking choice mechanism can be used instead - when making the switch, if there is the capacity to increase coding overhead, error correction performance can be kept at a similar level. If, on the other hand, coding overhead is a limiting factor, error correction ability will take a hit.

6.1.2 Integrating redundant information into the reserved bits

When considering the encoding scheme, one sees that the reserved bits cause a lot of coding overhead. It may seem wasteful, then, to pick the values of these reserved bits at random; an idea for improving the error-correction performance of the encoding scheme might be to make more intelligent decisions regarding the values of these bits. For example, would it be possible to integrate some redundant information into these reserved bits? Or set the reserved bits in such a way that parity information could be used during decoding?

In this work, several options for choice mechanisms which attempt to integrate redundant information have been suggested and assessed. These can be split into two categories: those which aim to improve the efficacy of Viterbi decoding only by their choice of reserved bits; and those which require a modified decoding algorithm to take redundant information into account.

Most Similar

The "`most similar`" choice mechanism is an entirely novel choice mechanism which was hoped to improve the error correction of the Viterbi algorithm. Its name comes from the fact that it tries to pick the reserved bit values to be as similar as possible to the input bits, the motivation for doing this is explained below. This choice mechanism comes with a requirement, however - that the number of reserved bits must be the same as the input size. In other words, half of a symbol must consist of reserved bits.

The Viterbi algorithm produces its maximum likelihood estimation for the decoded sequence by using a "cost" metric[1], which measures the difference between an observed symbol and the symbol that would have been expected assuming the FSM was in a particular state. It does this for every possible state at each observed symbol and accumulates the "cost" of the different possible sequences. The sequence with the lowest cost is then selected as the maximum likelihood sequence. This means that, for good error correction performance, the cost incurred from selecting an incorrect symbol should be as high as possible. In other words, the aim is to maximise the hamming distance between the different symbols which are used by the encoding[2].

In theory, the random choice mechanism is not optimal for maximising the Hamming distance between two symbols. For example, it is possible to imagine two inputs which differ by only a single bit. These two inputs would be appended to a set of reserved bits as described in section 4.2.3 and demonstrated in figure 4.4. Since these are selected at random from a list of valid candidates, it is possible that the set of reserved bits selected will be identical. This would mean that the two symbols only differ by one bit, and hence the Hamming distance between them is very

[1]The cost metric here is simply the Hamming distance.

[2]This is plausible since not all possible symbols that could be used by the encoding will be used - since the reserved bits form part of the symbol, the symbol which is used given a state and an input depends on the outcome of the choice mechanism.

little. This, in turn, may contribute to the Viterbi algorithm selecting the wrong sequence since these two symbols are very close to each other.

The "`most similar`" choice mechanism attempts to alleviate this problem by selecting the value of reserved bits which is `most similar` to the input bits. Take the above example, if two inputs differ by one bit, the `most similar` choice mechanism would attempt to pick a candidate from each input's set of valid reserved candidates. Provided the input bits themselves were in the reserved set, this would increase the difference between the symbols to two (since the symbol would now consist of the input bits repeated twice over). In theory, this should improve the cases when two similar inputs get the same reserved bit values. However, it comes with the trade-off that it will make worse the cases when two similar inputs receive very different reserved bit values.

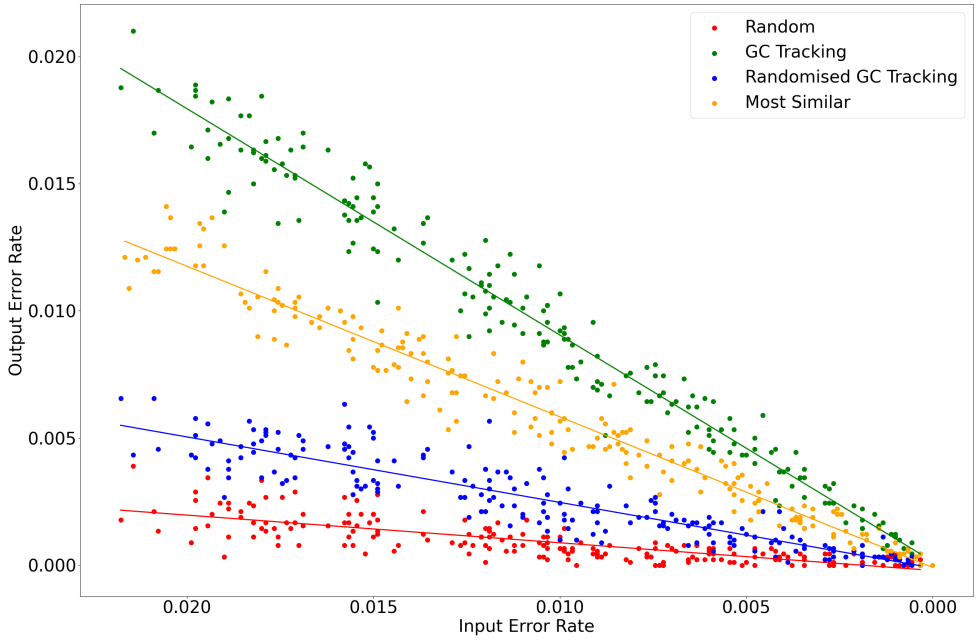


Figure 6.4: The `most similar` choice mechanism plotted against the three mechanisms already in use. The symbol size and number of reserved bits are all four.

The results of using the `most similar` choice mechanism are demonstrated in figure 6.4. This plot shows that, whilst better than GC-tracking, this mechanism has worse performance than both randomised GC-tracking and random choice. The most likely explanation for this is the first case discussed above (in which the `most similar` choice mechanism increases the difference between two symbols) is likely to be less common than the case in which `most similar` actually decreases the difference between two symbols. This is because, whilst random choice may assign identical reserved bits to similar inputs, it may also assign completely different reserved bits to similar inputs, thereby increasing the difference between the resulting symbols. However, `most similar` will always assign similar reserved bits to two similar inputs, even if it successfully avoids assigning identical reserved bits. Another potential downside of the "`most similar`" choice mechanism is that it seems likely to introduce or exacerbate STRs. As it was unsuccessful at reducing the error rate, the implementation of `most similar` is not included, however, it can be seen in appendix A.4.3.

Most Different

The "`most different`" choice mechanism follows the same logic as the `most similar` mechanism in trying to ensure that similar inputs don't randomly receive the same reserved bit values, but goes about it in the opposite way. By selecting the reserved bit values which are maximally different (have the largest Hamming distance) from the input, it tries to correlate the reserved bit values with the value of the input, thereby attempting to ensure that similar inputs are not assigned the same reserved bits. It suffers from the same drawback as `most different` in that the reserved bits and input must be of the same size, so once again, exactly half of any symbol must consist of reserved bits.

Advantages of the `most different` mechanism over the `most similar` mechanism include that it is not as vulnerable to STRs, and that since there are more ways for the reserved bit to be maximally different to the input string than to be identical to it, there are more optimal values to choose from. This means that two very similar inputs could receive reserved bit values which are quite different from one another.

Figure 6.5 plots the `most different` choice mechanism against the three choice mechanisms already in use, and once again, it is clear to see that the error correction is inferior to both random choice and randomised GC-tracking. This is probably for the same reason as the `most similar` choice mechanism. The implementation for the `most different` mechanism can be found in appendix A.4.4.

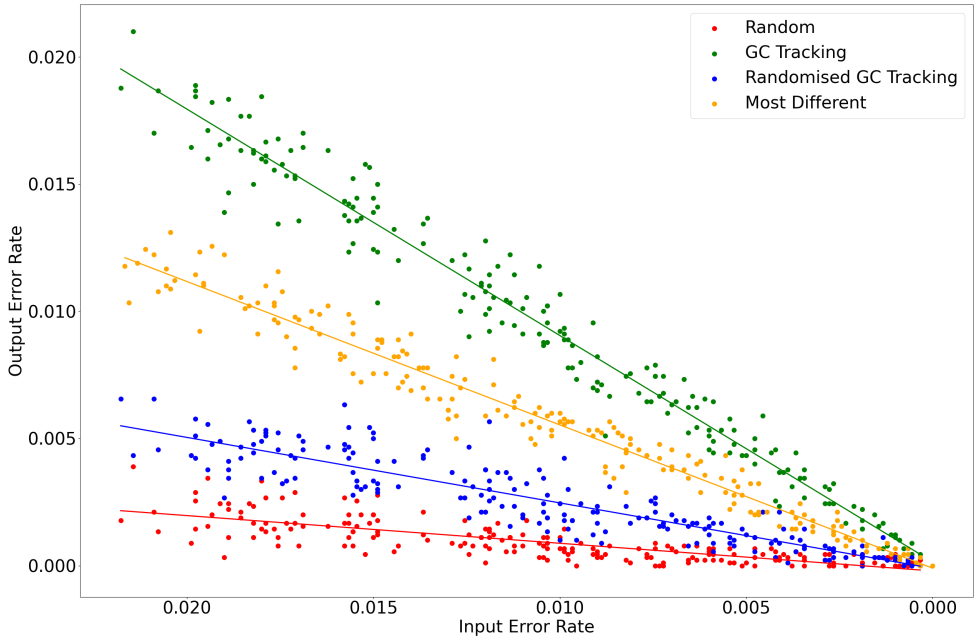


Figure 6.5: The `most different` choice mechanism plotted against the three mechanisms already in use. The symbol size and number of reserved bits are all four.

Random Unused

The previous two choice mechanisms tried to prevent two similar or identical inputs from receiving the same values for reserved bits in a programmatic way. A simpler method is to keep track of which reserved bit sequences have been used for which inputs and preferentially use reserved bits which have not been used with that input before. This choice mechanism is referred to as `random unused`, as it picks randomly between the reserved bit values which have not yet been used for a particular input. Only if there are no such values for the reserved bits does it reuse a value. The result of this should, in theory, be that the different symbols are used with a more even distribution. This is because a symbol consists of an input concatenated to reserved bits, and hence preferring to use a set of reserved bits which hasn't yet been used for a particular input should mean that symbols are repeatedly used for the same input fewer times. This should actually make the encoding scheme more similar to standard convolutional codes, which use different outputs for each state when faced with the same input (as can be seen in figure 2.4).

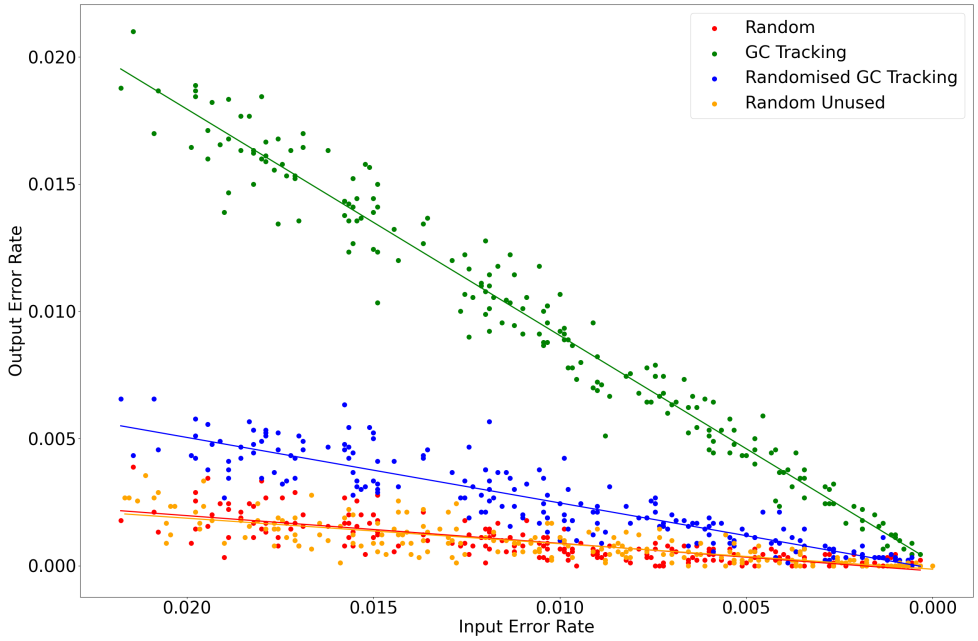


Figure 6.6: The `random unused` choice mechanism plotted against the three mechanisms already in use. The symbol size and number of reserved bits are all four.

To see if this method results in better error correction, its performance is graphed against the standard three

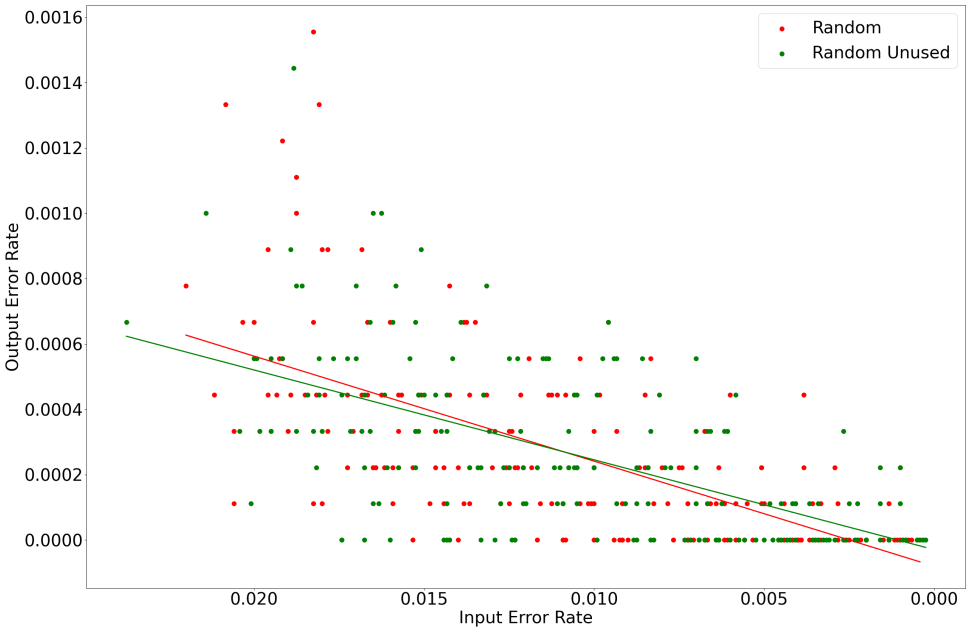


Figure 6.7: Comparison of `random unused` and `random` choice mechanisms with five reserved bits and a symbol size of four.

choice mechanisms in figure 6.6. Unsurprisingly, performance is very similar to the random choice mechanism, since the difference between these is subtle. To try and detect if any difference exists, figure 6.7 plots the performance of these choice mechanisms with an increased number of reserved bits (five), under the assumption that an increase in the number of reserved bits should amplify any difference in the efficacy of their selection.

Whilst figure 6.7 seems to demonstrate a very slight advantage gained by the `random unused` mechanism, this is not significant. Hence there is no evidence to suggest that `random unused` offers any improvement over random choice. The most likely explanation for this is that random choice already produces an even distribution of symbols, meaning the `random unused` algorithm offers little to no advantage. The implementation for this choice mechanism can be seen in appendix A.4.5.

Integrating Parity Checks

Since the previously mentioned choice mechanisms failed to provide any signification improvement over choosing the reserved bits randomly, a new strategy was attempted, which involved adding a new choice mechanism and making a corresponding change to the decoding algorithm.

This new choice mechanism was inspired by Hamming codes and other parity-based codes such as LDPC. Since the encoding mechanism does not have control to set the reserved bits directly (as it can only choose between candidates which do not violate the constraints), another method of integrating parity information into the encoding is attempted. This mechanism is called `alternating parity`, and it works by attempting to ensure that the parity of each consecutive symbol alternates, i.e. if one symbol has even parity, the next one will be odd, and the next will be even again, and so on. The idea behind this is to use this information in the decoding algorithm.

The choice mechanism works by filtering the reserved bit candidates, and only keeping those which guarantee the parity of the new symbol will be the opposite of the previous symbol. A random selection is then performed on the remaining candidates.

This information is then used in the decoding stage in the following way; at every observation (i.e. every symbol to be decoded), if two possible sequences have the same cost, the one with the opposite parity to the previous symbol is picked. The implementation of the choice mechanism, as well as the corresponding modifications made to the decoding algorithm, can be seen in appendix A.4.6.

As can be seen in figure 6.8, the `alternating parity` choice mechanism provides better error-correcting performance than randomised GC-tracking, however, this is not a fair comparison since it does not provide any of the GC variation control that randomised GC-tracking does. It fails to outperform the random choice mechanism, implying that any advantage gained by integrating the parity information was insufficient to offset the limitation placed by the choice mechanism on the number of reserved bit candidates.

6.1.3 Choice Mechanism Comparisons

Figure 6.9 plots all of the choice mechanisms on the same axis. From these results, it can be seen that the recommendations made regarding the selection between choice mechanisms remain mostly unchanged. If GC variation is an issue, randomised GC-tracking is recommended. If it is not, then the recommendation is random choice, either the standard version or the novel `random unused` mechanism. The selection between the two flavours of random choice

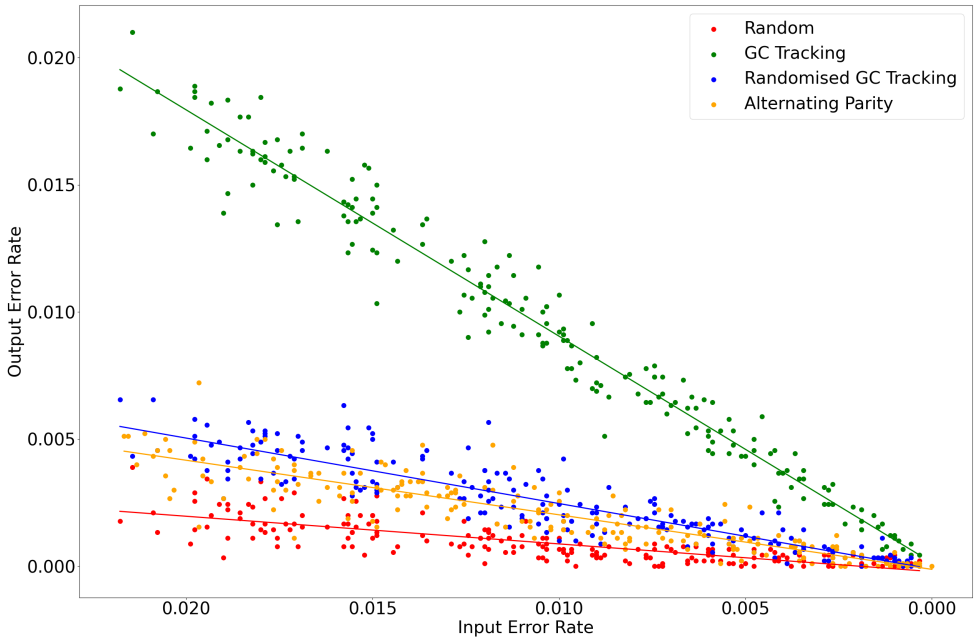


Figure 6.8: The `alternating parity` choice mechanism plotted against the three mechanisms already in use. The symbol size and number of reserved bits are all four.

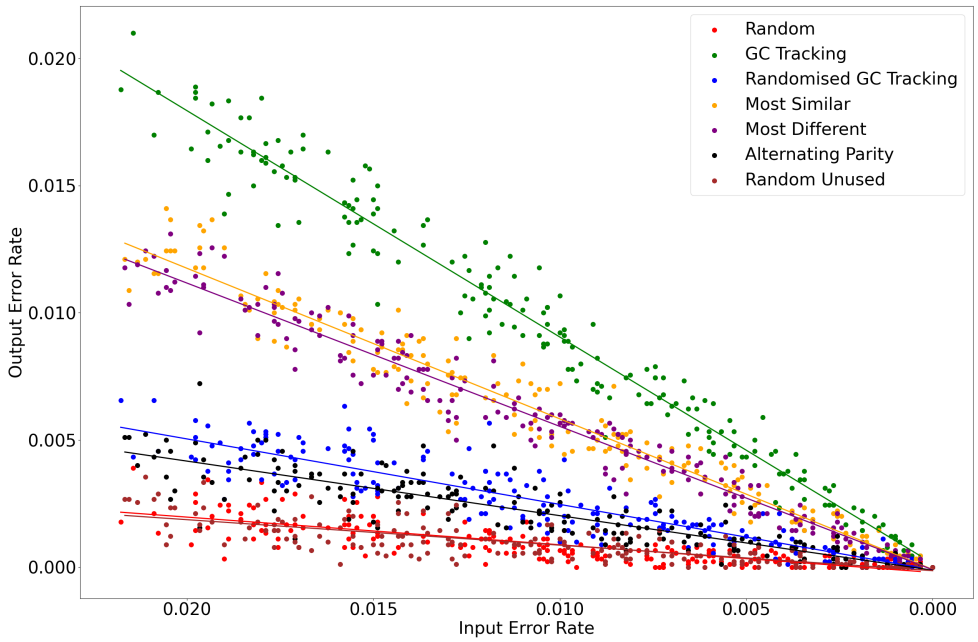


Figure 6.9: The error correction of all choice mechanisms. The symbol size and number of reserved bits are both four. The `most similar` and `most different` choice mechanisms both result in worse error correction than random and randomised GC-tracking. The `alternating parity` method improves upon the performance of randomised GC-tracking, but without providing any of the GC variation control, so it is inferior to random choice. The `random unused` mechanism seems to provide the best error correction out of all of them, but its difference with random choice is insignificant.

is unlikely to make a difference, so the only important consideration is that the standard version will be faster (though only the speed of FSM generation is affected, not that of encoding/decoding), but the `random unused` version may offer a very slight improvement in error correction and also perhaps be more stable (less prone to degradation in performance due to suboptimal random choices for the values of the reserved bits).

6.2 Constraint-based decoding

The constraint set is used to construct the finite state machine, which is then used during encoding and decoding. Since the FSM contains all the information needed by the convolutional coder, the constraint set is not required

during encoding/decoding. During decoding, the Viterbi algorithm makes decisions about which sequences are more or less likely given a set of observations. These "decisions" are based on the Hamming distance between the output expected from a given combination of state and input and the observation received by the algorithm. This system leaves certain soft information unused; it is guaranteed that the encoded DNA sequence avoids certain symbol combinations to comply with the specified constraints. This provokes a question - "might it be possible to utilise this constraint information during decoding to improve error correction?". This question is investigated below.

To see if the constraint list could be leveraged during the decoding stage to improve error correction, a representation of the constraint list was added to the convolutional coding / Viterbi decoding rust package. This, unfortunately, increases the coupling between the two rust packages. During decoding, each new observation was checked against all the possible states which could have led to that observation. Any combination of state and observation which breaks a constraint on the constraint list can then be discarded since it is impossible that the encoding scheme would have encoded such a combination. In practice, this "discarding" was done by adding a very high cost, referred to as a "penalty", to any path which broke a constraint, which ensured that it would be disregarded provided there was any other reasonable path which did not break a constraint. The implementation of constraint-based decoding can be viewed in appendix A.5.

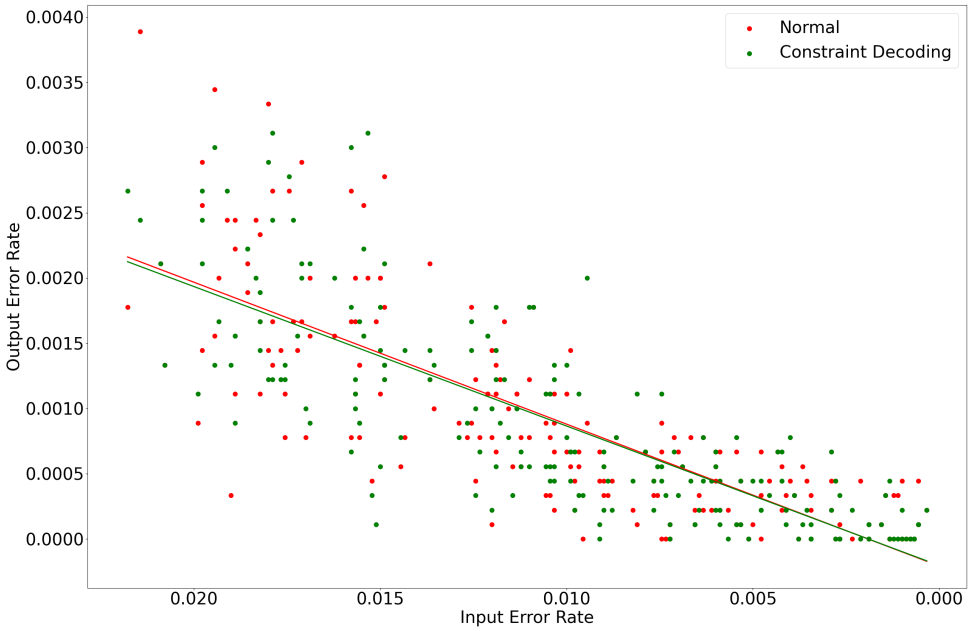


Figure 6.10: Comparison of Constraint-based decoding and standard Viterbi decoding. The symbol size and number of reserved bit are both four.

Figure 6.10 shows the results of this experiment. Clearly, there is no additional advantage gained by taking constraint information into account during decoding. This indicates that the majority of errors which are not successfully corrected by the encoding scheme do not lead to a constraint violation (probably because they are single base flips), so taking the constraints into account does not help in correcting them.

A further experiment was carried out to determine whether constraint-based decoding could be used to correct burst errors. The logic behind this was that errors in bursts are more likely to cause constraint violations because they flip several bases in a single symbol or concatenation site of symbols, hence they can have a greater effect on the GC content or homopolymer length within a single symbol. However, no improvement in burst error correction was observed when using constraint-based decoding as compared to standard Viterbi decoding. These results are omitted for brevity but can be found in appendix A.5.

6.3 A soft constraint for mitigating short-tandem repeats

STRs, which are consecutive repetitions of a subsequence, are not addressed in any way by the original encoding scheme as it is described in [13]. There is a good reason for this - it is not possible to entirely mitigate this constraint with an FSM-based encoding unless either, the symbol size is very long, or the state space is expanded to include multiple concatenated symbols. Neither of these options is desirable as both would make encoding and decoding much slower, and the latter would also make FSM generation more complicated. Therefore, the solution proposed does not entirely mitigate violations to this constraint, but can drastically decrease them.

The solution works by adding a "soft constraint" to the `Constraints` object. The modified version of `Constraints` can be viewed in appendix A.6.2. This soft constraint is added in the form of two new attributes: `str_lower` and `str_upper`, which represent the lower and upper bounds for the lengths of the STRs which should be reduced. What is meant by "STR length" in this case is the number of nucleotides in the subsequence which is repeated. For example, an "ACGTACGT" would have an STR length of four, whereas "ACGACGACG" would have an STR length of three. They

are not named min and max to avoid confusion with other constraints such as `gc_min`, `gc_max` and `max_run_length`, each of which refer to the minimum/maximum rate or number allowed.

A limitation of this mechanism is that `str_upper` must be at most equal to the symbol size of the encoding. This is due to the fact that each concatenation site of two symbols is inspected independently during FSM construction, meaning the longest subsequence which can be checked for STRs is twice the length of the symbol size. To avoid this limitation, the concatenation sites of three or more symbols would have to be inspected at the same time, and the structure of the FSM would need to be altered so that states represented not just one but the two previous symbols. This would make both finite-state machine generation more complicated, and make encoding/decoding slower, whilst only increasing the length of the longest STR that can be detected by 50%.

Algorithm 4 Soft STR Constraint Check

Input: sequence: string, str_lower: int, str_upper: int
Output: str_found: bool

```
Require: str_llower ≤ str_lupper
  for size ← str_llower, str_lupper do
    for i ← 0, sequence.len - (size × 2) do
      one ← sequence[i : i + size]
      two ← sequence[i + size : i + (size × 2)]
      if one == two then
        return True
      end if
    end for
  end for
return False
```

The mechanism works in a simple way. The `satisfied` method of the `constraint` object is modified to check for STRs within the concatenation site of two symbols. It does this by iteratively checking for STRs of each length in the range [`str_lower`, `str_upper`] inclusive. The code used to check for the presence of STRs is shown in algorithm 4 - if this function returns true, the concatenation being inspected is considered invalid, and therefore the `satisfied` method will return false. The Rust implementation can be found in appendix A.6.3. This algorithm will eliminate many but not all STRs of a particular length; the ones which are missed are those which cannot be detected within the concatenation site of two symbols, i.e. those for which a single repetition spans at least three symbols. Table 6.1 shows various examples of STRs of length three with a symbol size of four to demonstrate the cases in which STRs can and cannot be prevented.

Sequence	S1	S2	S3	Detected & Prevented?
ACGACGTATGCG	**ACGA**	**CG**TA	TGCG	Prevented by disallowing S1 + S2
G**ACGACG**TATGC	G**ACG**	**ACG**T	ATGC	Prevented by disallowing S1 + S2
CG**ACGACG**TATG	CG**AC**	**GACG**	TATG	Prevented by disallowing S1 + S2
GCG**ACGACG**TAT	GCG**A**	**CGAC**	**G**TAT	Not detected as no two symbols contain an STR
TGCG**ACGACG**TA	TGCG	**ACGA**	**CG**TA	Prevented by disallowing S2 + S3
ATGCG**ACGACG**T	ATGC	G**ACG**	**ACG**T	Prevented by disallowing S2 + S3
TATGCG**ACGACG**	TATG	CG**AC**	**GACG**	Prevented by disallowing S2 + S3

Table 6.1: Each of the sequences shown is a rotation of the first sequence, which contains the STR "ACGACG". Assuming a symbol length of four, the sequence will be spread out across three symbols. Since the STR checking happens at the concatenation site of two symbols, if the inspection of three consecutive symbols is required to detect the STR, it will not be detected. Therefore some STRs will get through.

Having established that this method should eliminate many STRs, a method for measuring STR prevalence is defined. This is done by sweeping an entire DNA sequence and checking for STRs of a given length. The function used to do this can be seen in appendix A.6.4. Figure 6.11 shows the results of such a sweep. This figure plots the number of STRs divided by sequence length (dividing it by the sequence length is necessary as the hyperparameters affect the length of the encoded string). These results show that the (original) encoding scheme has no effect on STRs since the STR profile of the unencoded string is almost identical to the encoded strings. It additionally shows that the various hyperparameter choices, such as the number of reserved bits and the symbol size, do not matter. Another interesting observation from this figure is that long STRs are much less likely to occur than shorter ones. This suggests that the limitation on STR length may not pose too much of a problem.

Next, the effect on STRs of the soft STR constraint is assessed. The results of this assessment can be seen in 6.12, in which a symbol size of five is used. Since the symbol size used is five, STRs longer than this cannot be addressed. This can be seen in the figure by observing that the only decrease occurs in STRs of length five or less. This still gives good results, however, since short STRs are much more common than long ones.

Previously, no relationship was found between symbol size and STR prevalence. However, now that the soft STR constraint is being used, a relationship between these factors should be expected, since the symbol size limits the length of the STRs which can be detected. Figure 6.13 plots this relationship. As expected, the longer the symbol size, the less the prevalence of STRs. This occurs for two reasons. The first is the one already mentioned - that STRs

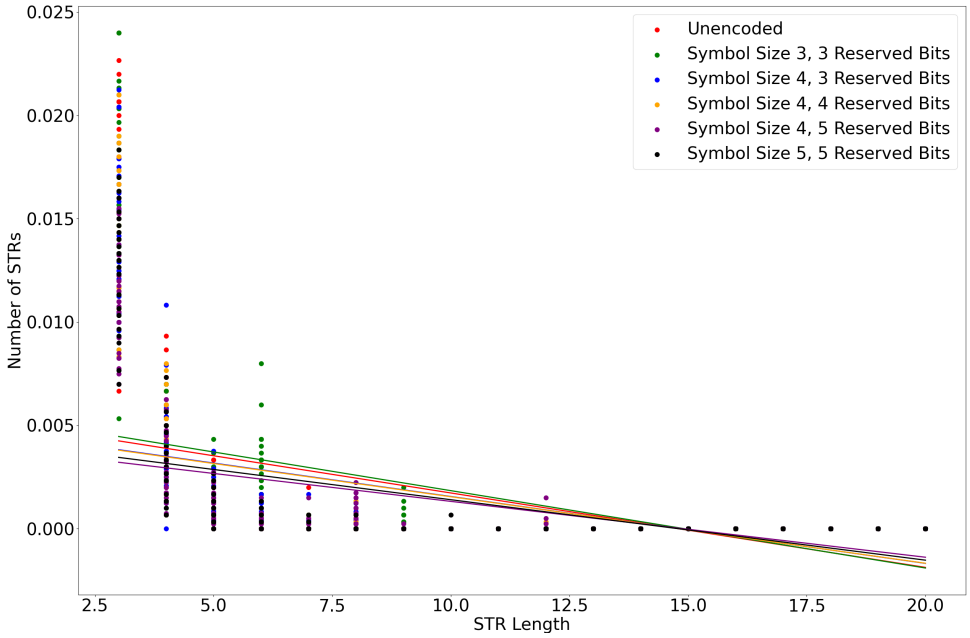


Figure 6.11: The number of STRs divided by the length of the sequence. Demonstrates that there is no effect on STR prevalence due to the encoding regardless of hyperparameter selection.

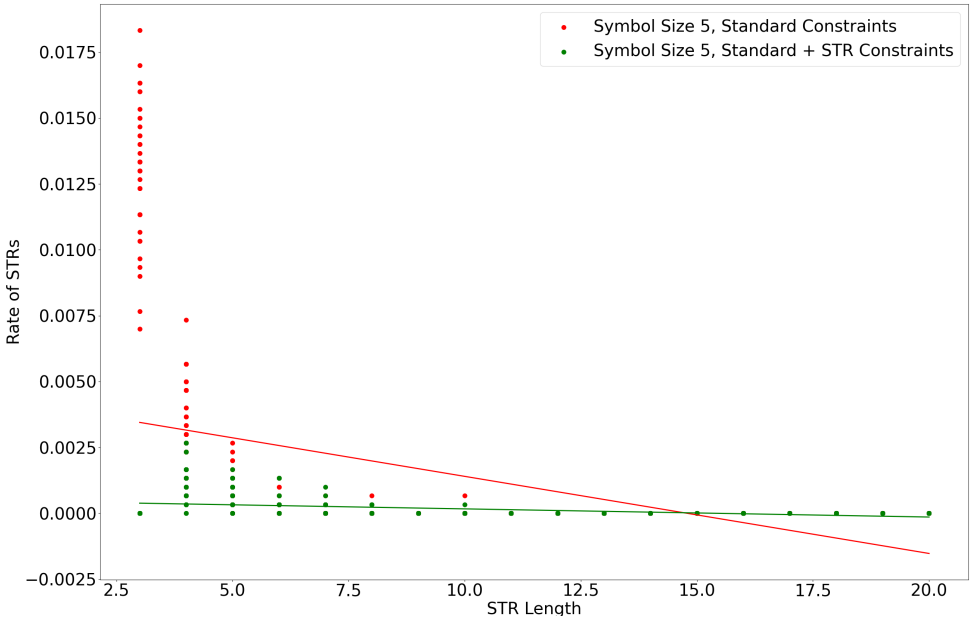


Figure 6.12: The effect of the soft STR constraint on the prevalence of STRs. The symbol size and number of reserved bits are both set to five.

longer than the symbol length cannot be detected. However, an inspection of figure 6.13 reveals that a larger symbol size also results in fewer short STRs. The reason for this is simple - the smaller the STR length compared to the symbol size, the fewer STRs will be missed. For example, with a symbol size of five, it becomes impossible to miss any STRs of length three since STRs of length three can always be detected at the concatenation site of two symbols (i.e. the situation demonstrated in table 6.1 becomes impossible). This is evidenced by the fact that all the data points for a symbol size of five and an STR length of three are at zero in figure 6.13.

Since this additional constraint has the effect of reducing the number of valid reserved bit candidates (by eliminating those which lead to STRs), it is reasonable to expect the error-correction performance to take a hit. However, figure 6.14 demonstrates that there is little to no effect on error correction as a result of the soft STR constraint, regardless of symbol size. This may be explained by the fact that symbol concatenations containing STRs are relatively rare already, and so the number of valid reserved bit candidates does not change by much.

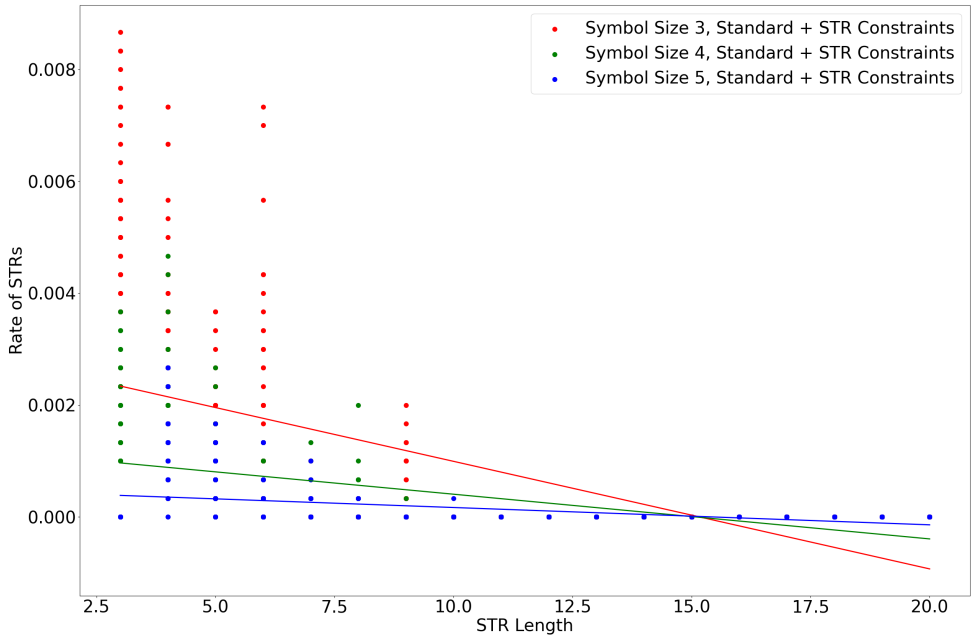


Figure 6.13: The effect of the soft STR constraint on the prevalence of STRs.

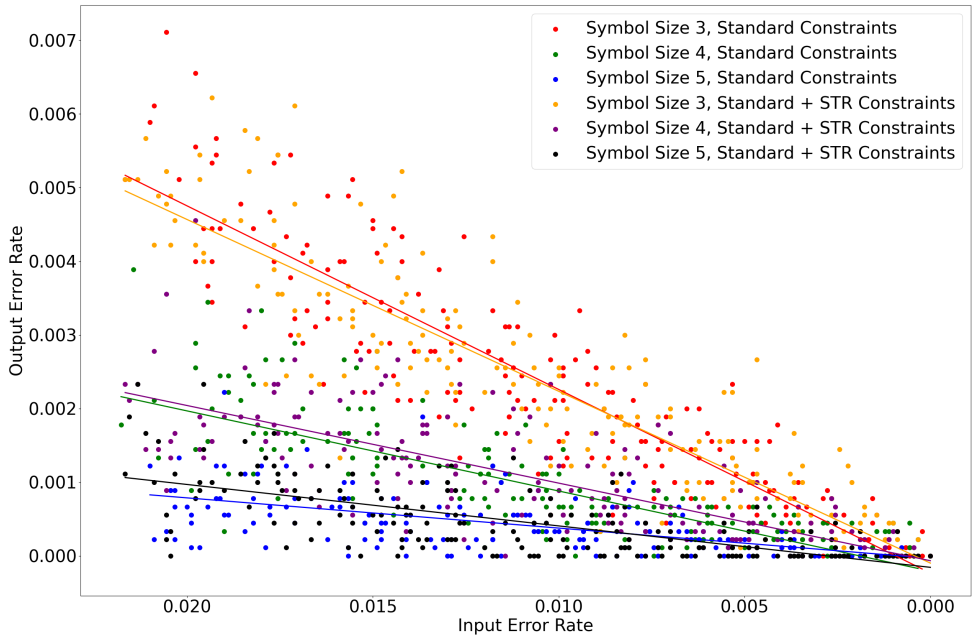


Figure 6.14: The effect of the soft STR constraint on error correction.

6.4 Conclusions regarding the extensions

In setting out to extend the encoding schemes, the three aims were: to marry the error correction of the random choice mechanism with the GC variation control of the GC-tracking mechanism; to improve upon the error-correction ability of the random choice mechanism; and to partially or entirely mitigate STRs. The first of these was achieved through the introduction of a novel choice mechanism called randomised GC-tracking, which maintains all the GC control of GC-tracking whilst recovering most of the error correction of random choice. The second aim, however, proved more elusive and was not achieved despite experimentation with several different choice mechanisms and modifications to the decoding algorithm. The third aim was achieved through the introduction of a "soft" STR constraint. The extensions to the encoding scheme are therefore considered to be a partial success. Future work could be carried out to find ways of improving upon the error correction of the random choice mechanism.

Chapter 7

Evaluation

There has already been a great deal of evaluation throughout this work. In essence, chapter 5 is dedicated to evaluating the encoding generation scheme, and chapter 6 contains evaluations of every extension which is proposed. Section 4.3 evaluates the extent of performance improvements gained by refactoring Python code into Rust. In this chapter, the previous evaluations will be summarised, but in addition, some further evaluations will be presented. In particular, the following factors will be discussed:

- The extent and significance of the novel results produced in this work (Section 7.1).
- The error correction capabilities of the encoding scheme as compared to a similar ECC with the same overhead (Section 7.2).
- The level of constraint compliance achieved by the encoding scheme (Section 7.3).
- The significance of enhancements made to the encoding scheme in this project (Section 7.4).
- The performance properties of the encoding scheme (Section 7.5).
- The trade-offs and limitations of the encoding scheme (Section 7.6).
- The software implementation of the encoding scheme (Section 7.7).

This evaluation reveals some limitations to the encoding scheme as it is presented in this work, leading to items of future work which are enumerated along with the conclusion in chapter 8.

7.1 Novel Elements

This project has contributed a number of novel results, some novel algorithms, and new implementations of existing algorithms. The most important novel result presented in this work is that the encodings created by the encoding generation scheme can be used as error-correcting codes whilst continuing to serve their original purpose of ensuring compliance with biochemical constraints. In chapter 5, it was clearly demonstrated that provided the reserved bit values are picked randomly from the set of valid candidates, the resulting candidates can correct substitution errors. This is a small but significant novel result, as it proves that a simple encoding scheme can simultaneously avoid constraint violations and correct errors. The key point here is that the coding overhead (the reserved bits) is being used for both error correction and constraint compliance. Sections 7.2 and 7.3 evaluate how the reserved bits are "shared" between these two applications, and demonstrate that each reserved bit is serving a dual purpose. It must be stated that this contribution is modest - the error correction capabilities of the encodings do not extend to insertion and deletion errors nor long bursts of errors. However, since the primary aim of the encodings is to avoid constraint violations, it can be said that some error correction is essentially provided for "free".

Further experiments were carried out to demonstrate how best to configure the encoding generation scheme, and how to navigate the various trade-offs between the configurations. The results of these experiments led to a set of recommendations, also entirely novel, of how best to configure the encoding scheme given different priorities. These recommendations are presented in sections 5.2.4 and 6.1.1 and represent a small contribution which may serve to guide future work which makes use of the encoding scheme. Future researchers may use the recommended configurations to avoid experimentation with different configurations altogether, or they may use them as starting points for their own experiments. This contribution is a small one since many of the recommendations are somewhat obvious; increasing coding overhead improves error correction, larger symbol sizes result in lower error rates but at the cost of slower decoding, etc. However, some of the recommendations are less obvious and, therefore, more useful; the fact that GC-tracking can be used to comply with GC-variation constraints but has no ability to correct errors, the fact that randomised GC-tracking has error correction performance which lies between GC-tracking and random choice, etc.

Several novel algorithms were presented in this work, but the two most significant are likely to be randomised GC-tracking and "soft" STR constraints. These are simple algorithms and modest contributions, but the result is that the encoding scheme can also comply with completely with GC-variation constraints and partially with STR

constraints. This extension to the encoding scheme means that it can avoid violations of all the major biochemical constraints stipulated by DNA manufacturers whilst also correcting substitution errors.

A further contribution is the implementation of the encoding scheme, including rust libraries for FSM generation and convolutional coding / Viterbi decoding. The Rust library for convolutional coding can, in principle, be used by anyone looking for an efficient implementation of convolutional codes and Viterbi. This software is provided freely and openly under an MIT licence for use by anyone for any purpose. The performance of the software is evaluated in section 7.5, and its design and maintainability are evaluated in section 7.7.

Each of these is a modest contribution, but together they serve to produce a usable encoding scheme which can adjust for the changing requirements of DNA synthesis and sequencing technologies whilst correcting substitution errors and preventing violations of biochemical constraints.

7.2 Comparison with standard convolutional codes

The encoding generation scheme presented here simultaneously enforces given biochemical constraints and also provides protection against substitution errors. However, it is unclear how much of the overhead is being used for constraint compliance and how much is being used for error correction. To assess this, a comparison is provided between a standard $1/2$ convolutional code and an encoding generated by the encoding scheme with the same overhead.

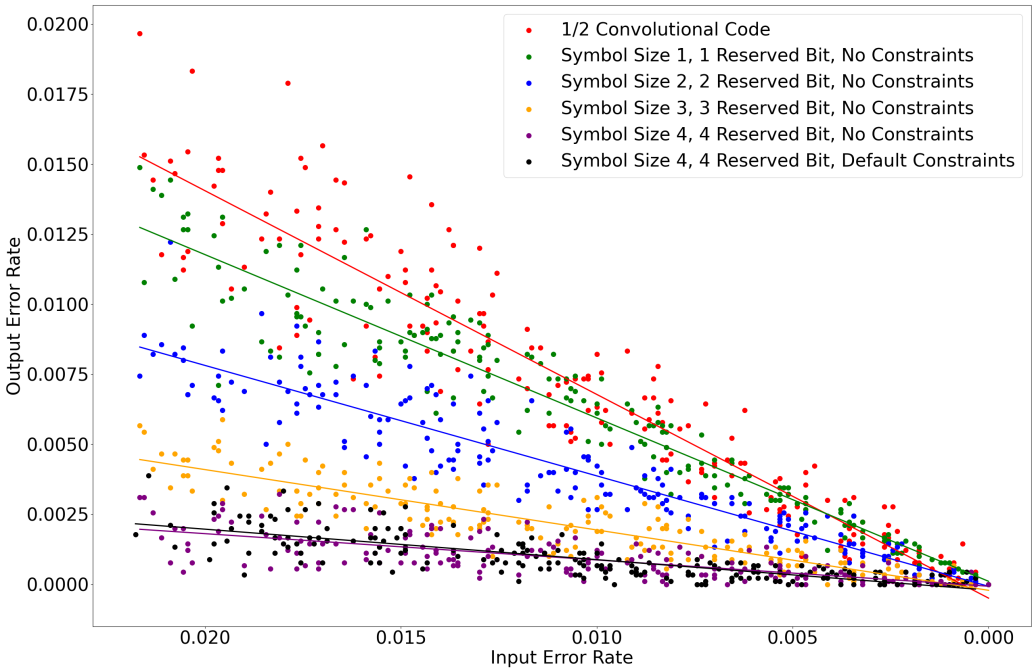


Figure 7.1: Comparison of $1/2$ convolutional code with encodings generated by encoding scheme. In all cases, exactly half the encoded sequence is used for error correction, so coding overhead is constant.

Figure 7.1 shows the results of this comparison. Interestingly, a symbol size of one with one reserved bit seems to outperform the $1/2$ convolutional code by a small margin. This is surprising due to how similar these encodings are in structure - the only difference is likely to be the random allocation of the reserved bit, which may have produced a trellis which is more effective at correcting nucleotide errors. As expected, longer symbol sizes greatly improve error correction, but at the cost of slower encoding/decoding. These results suggest that the encoding scheme provides superior error correction to the $1/2$ convolutional code despite having the same overhead. Another thing to note is that there seems to be very little difference between the error correction performance with no constraints and with the default constraint set, which indicates that avoiding violations to the default set of biochemical constraints does not degrade error correction.

7.3 Constraint compliance

The encoding generation scheme has two aims - to avoid violating commonly specified biochemical constraints and to do so whilst correcting errors. The original encoding scheme, as described in [13], was able to guarantee compliance with GC limits, maximum homopolymer length and reserved subsequence constraints. The enhanced version of the encoding scheme can also mitigate GC variation and some STR constraints. The extent to which the enhanced encoding is able to mitigate these constraints has already been evaluated - in section 6.1.1 for GC-variation and section 6.3 for STRs.

To further evaluate constraint compliance, IDT's online `gBlocks` tool[1] is used. To carry out this evaluation, a particularly pathological DNA sequence is constructed manually. This DNA sequence contains several long homopolymer runs, several STRs, sections of very high and very low GC content, and consequently, a very high GC variation. This pathological sequence is then entered into the IDT tool, which rejects it and gives it a very high complexity score of 142.3, as can be seen in figure C.1[2]. Next, the pathological sequence is mapped to a bit string, yielding the bit sequence which would have produced such a sequence without the encoding scheme. The default constraints are then used to create an FSM with symbol size and reserved bits set to four. The bit string is then encoded using the FSM, and the encoded string is mapped back into a DNA sequence. Finally, the new DNA sequence, which represents the same information as the pathological one but is twice as long, is given back to the IDT online tool. The encoded version of the sequence is accepted by the tool, with a greatly reduced complexity score of 7.1, which can be seen in figure C.2.

The difference in the complexity scores between the encoded and unencoded sequence suggests that the encoding is highly effective at avoiding constraint violations. In reality, sequences as pathological as the one tested are likely to be very rare, so the difference in complexity will not usually be this great.

7.3.1 The cost of compliance

An interesting question is exactly how the reserved bits are "shared" between error correction and constraint compliance. Curiously, figures 5.7 and 7.1 both contain experiment iterations in which the only factor varied is the strictness of certain constraints. In both of these cases, there appears to be little to no difference in error-correcting performance caused by increasing the strictness of the constraints. This may seem surprising at first, but it is explained by the fact that the default constraints, whilst sufficient to comply with the constraints of most DNA manufacturers, are not too limiting, so there are still many valid reserved bit candidates to choose from, and so the strictness of the constraints does not act as a limiting factor for the error correction. This, in essence, means that every reserved bit is being used for both error correction and constraint compliance.

7.4 Improvements made to the original scheme

Chapter 6 is dedicated to various attempts at improving the original encoding scheme as described in [13]. The chapter also contains the evaluations of these improvements with respect to either error correction or constraint compliance. These evaluations show that the attempts to improve upon the error correction of the original scheme were unsuccessful, yielding either worse results or no difference. However, two novel algorithms, randomised GC-tracking and soft STR compliance, were both successful in their aims of improving constraint compliance whilst maintaining error correction.

In evaluating these new mechanisms, it is important to note their limitations - neither of these algorithms can entirely prevent violations of their target constraints. As shown in figure 6.3, randomised GC-tracking decreases GC-variation, but whether this decrease is sufficient depends on the strictness of a manufacturer's constraints. Similarly, figure 6.12 shows a decrease in STRs due to the soft STR constraint, however, whether this is sufficient or not once again depends on the DNA manufacturer. These figures clearly show that both mechanisms work to mitigate violations of their target constraint, even if not entirely.

Since neither mechanism requires any further coding overhead, they, in some sense, come for "free". However, randomised GC-tracking does result in worse error correction, as shown in 6.1, and so there is a trade-off to be considered there. The soft STR constraint does not seem to affect error correction as demonstrated by figure 6.14, and so it is included in the default constraint set.

7.5 System performance

The changes in performance due to switching from Python to Rust have already been evaluated in section 4.3. In this section, the performance of the system is evaluated as a whole, with respect to the two most important factors affecting the performance. These two factors are symbol size and sequence length (i.e. the length of the sequence to be encoded). The encoding scheme consists of three primary sections: FSM generation, convolutional coding, and Viterbi decoding. Profiles of each of these can be found for both Python and Rust in appendix B, here profile of the "complete" system is presented, which involves one FSM generation operation, followed by a convolutional coding and Viterbi decoding operation.

Figure 7.2 shows the relationship between run duration and sequence length, which is linear. This result is to be expected since FSM generation has constant time complexity with respect to sequence length, whereas convolutional coding and Viterbi decoding are both linear in terms of sequence length. This result demonstrates that much longer sequence lengths could be encoded/decoded before this becomes a limiting factor; even the longer sequence lengths of over 1000 nucleotides took less than a second to encode and decode. Figure 7.3 demonstrates the relationship between system performance and symbol length, which is exponential. Again, this is expected since FSM generation and Viterbi decoding both have exponential time complexity with respect to symbol length. This demonstrates that symbol length will be a limiting factor for performance if it is pushed beyond six.

[1] Available at: https://eu.idtdna.com/site/order/gblockentry

[2] Moved to the appendix for brevity.

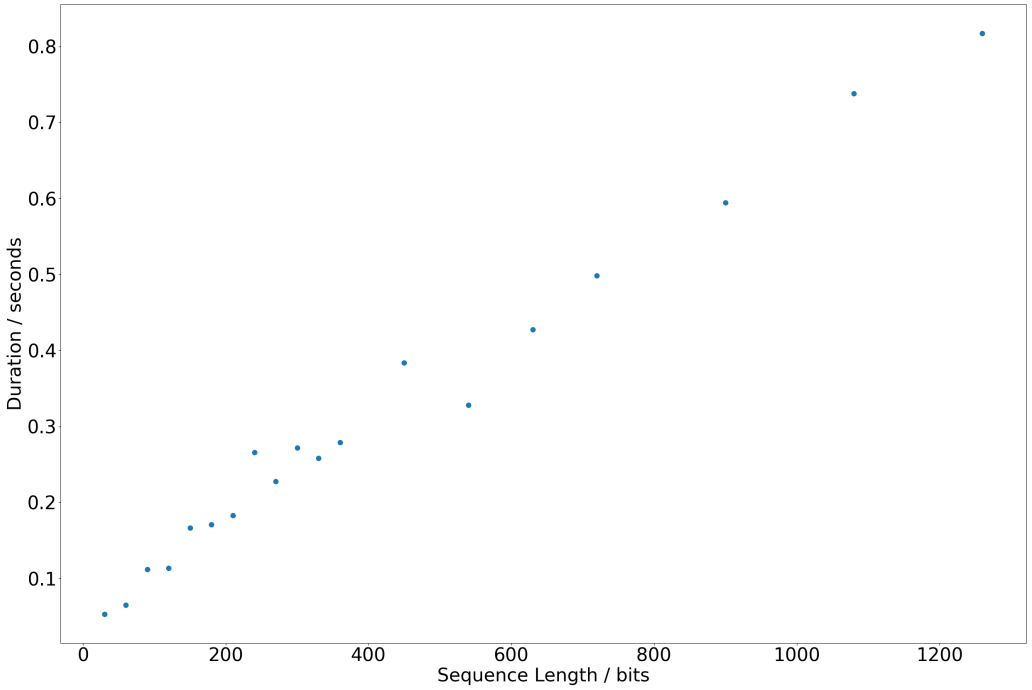


Figure 7.2: A profile of the whole system with respect to the sequence length.

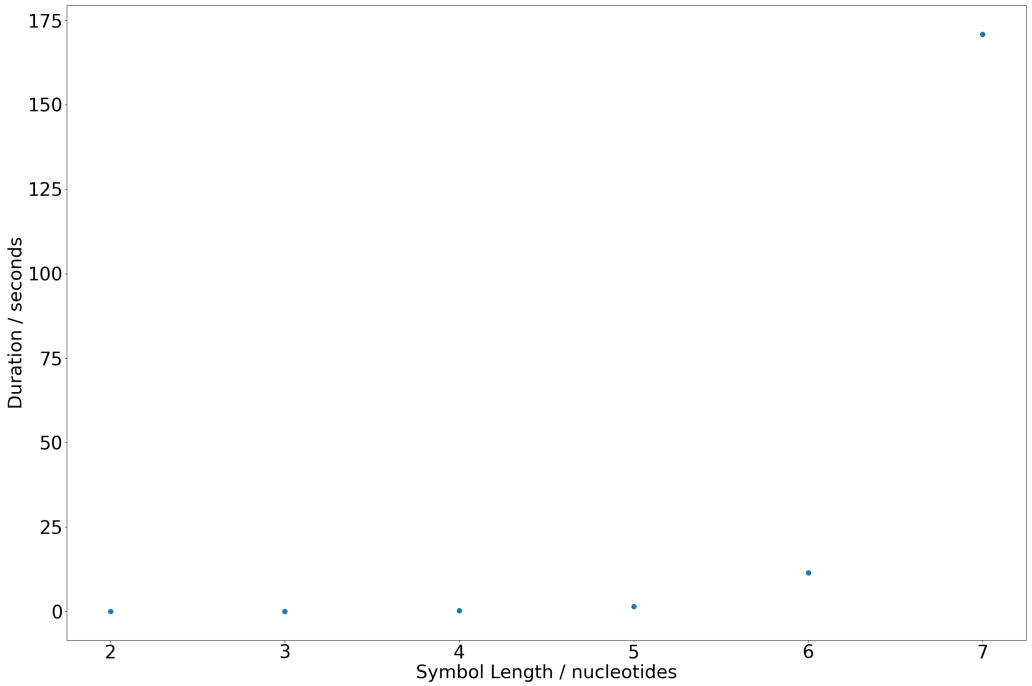


Figure 7.3: A profile of the whole system with respect to symbol size.

7.6 Trade-offs and limitations

There are a number of important trade-offs and limitations which must be mentioned when evaluating the encoding scheme. The limitations with regard to constraint compliance include that STRs are only reduced and not eliminated and that STRs longer than the symbol length cannot be reduced at all. The limitations with regard to error correction are that burst errors, as well as insertions and deletions, are not corrected by the encoding scheme in its current form. Another limitation with respect to error correction is that the error rate seems, based on all the results in chapter 5, at best, to be reduced between one and two orders of magnitude. For many sequences, this is sufficient to eliminate all substitution errors entirely, however, for other sequences, a number of errors get through. These errors would either need to be corrected through majority voting of a number of sequences, or through the integration of an additional error-correcting code. Thus, if the encoding scheme is used as the only ECC, sequencing costs are likely to be kept higher by the need to sequence many identical strands to perform a majority vote. Equally, if another ECC

was integrated, this would also increase both synthesis and sequencing costs due to an increase in overhead.

The various trade-offs which must be made when deciding upon the configuration of the encoding scheme have already been discussed at length, but they can be summarised simply. Increasing the symbol size improves error correction and STR mitigation but comes at the cost of an exponential slowdown. Increasing the number of reserved bits improves error correction and enables compliance with a stricter set of constraints, but increases coding overhead and, therefore, costs. Switching from using the random choice mechanism to using randomised GC-tracking decreases GC variation but degrades error correction. These are the main three trade-offs that must be taken into account when configuring an encoding, and error correction and decoding speed are both heavily affected by how these trade-offs are handled.

7.7 Software Evaluation

Software is not the primary focus of this work, but simply a means to an end. However, it may still be useful to review and evaluate the software for the purpose of guiding future development. In terms of maintainability, parts of the software implementation fare better than others. For example, the two Rust packages are relatively self-contained and are somewhat decoupled from each other and the Python code. This is advantageous both for testing and maintenance. However, when it comes to the Python code, the architecture is less well thought-out, with a chaotic dependency structure. This is the result of purposefully taking out "technical debt" to allow for faster development time. Therefore, it is crucial, if the implementation of the encoding scheme contributed by this project is to continue to be used, that refactoring and bug fixing be carried out.

If the encoding were to go into production as a product to be leveraged by researchers and companies, more attention would need to be paid to improving performance and also providing more useful error messages - the code currently contains several `assert` statements which should be removed if it were to become a professional application. To fulfil the aim of improving performance, one approach would be to rewrite the sections which are still in Python into Rust. A second approach would be to rewrite the various algorithms, such as FSM generation, convolutional coding and Viterbi decoding, in a concurrent/parallel manner. Currently, all the algorithms are implemented sequentially, when at least some of them could be parallelised.

Despite these limitations, the software implementation has been fast and maintainable enough for use throughout this work and is sufficient for research purposes.

Chapter 8

Conclusion and Future Work

This work has demonstrated the efficacy of the encoding generation scheme described by Sella *et al.* in [13] as an ECC. It has clearly shown that encodings generated by the scheme are effective for correcting substitution errors in DNA storage, which is a novel result. The limitations of the encoding scheme's error correction with regard to insertions, deletions and burst errors have also been shown. Furthermore, different hyperparameter configurations which can be used to generate encodings have been assessed, and recommendations have been made regarding which configurations are suitable depending on the priorities and limitations of a user. It has also been demonstrated that error correction can be achieved by encoding without compromising compliance with biochemical constraints. Novel enhancements have been proposed to improve the constraint compliance behaviour of the encoding scheme, and then evaluated with respect to both their target constraints and their effect on error correction. Finally, an efficient implementation of the extended encoding scheme has also been contributed.

Together, these contributions result in an encoding scheme which can correct substitution errors whilst complying with all the major biochemical constraints stipulated by DNA manufacturers.

Future Work

The revelation that the Sella encoding as described in [13] can be extended to act as an error-correcting code opens the door to many new possibilities, which can be explored in future work. Some of the most pressing questions are examined below.

8.1 Integrate other ECCs

Most state-of-the-art approaches to DNA storage include one encoding step for addressing constraints and another for error correction. The encoding scheme described in this work is able to address both of these aims, however, it is not equally effective against all types of errors. Therefore, further work could be done to ascertain if dedicated error-correcting codes could be integrated into this encoding scheme to improve its error correction, especially with regard to insertion and deletion errors. It would be interesting to compare which types of ECC (Reed Solomon, LDPC, etc.) are most effective for reducing the error rate whilst minimising additional overhead. An approach which seems likely to yield good results is to keep the convolutional encoding scheme as the only inner code and add an outer code for better insertion/deletion performance.

8.2 Make better use of the reserved bits

The current encoding scheme makes use of a number of reserved bits per symbol. Further work could be done to find better ways of selecting the values of these reserved bits. Some of the selection methods included in this project are random choice, GC-tracking and randomised GC-tracking. It has been demonstrated that the method for selecting reserved bits makes an enormous impact on the error-correction ability of the encoding, so it seems likely that there are further ways to decrease the error rate by varying this parameter. In chapter 6, the attempt was made to include redundant information in the reserved bits, and then to make use of that information during decoding. Unfortunately, none of the attempts demonstrated better error correction performance than the random choice mechanism. However, it may still be possible to select the reserved bits in a more considered way, such that some error-correcting information is stored in the reserved bits, and subsequently made use of during decoding.

8.3 Insertions, deletions and burst errors

Prior work on convolutional codes has established that certain alterations to the Viterbi algorithm be made to allow for the correction of deletion errors [46]. It may be possible to integrate this enhanced version of the Viterbi algorithm into the encoding generation scheme, which may allow it to correct some deletion errors without the need for an outer code. This report also hypothesises that an increase in symbol length will result in better handling of burst errors. In general, a very interesting avenue for further research could be to investigate how the encoding scheme

can be leveraged to correct insertions, deletions and burst errors without the addition of another error-correcting code.

8.4 Weighted connectivity matrix

A possible piece of future work is mentioned by Sella *et al.* - FSM construction could be done with a connectivity matrix of cost violations rather than a binary connectivity matrix. Rather than each concatenation being marked as either valid or invalid, each concatenation would be assigned a violation cost, representing the degree to which it violates the constraints. This would have an interesting interplay with the choice mechanism. In the simplest case, the choice for reserved bits with the lowest violation cost could be picked, however, it is possible to imagine a more advanced choice mechanism which may occasionally pick a higher cost violation in return for better error correction.

8.5 Further investigations into the effect of symbol length

In this work it was demonstrated that an increasing the symbol length used by the encoding generation scheme improves the error correction performance without increasing the coding overhead. Due to the exponential time complexity of the Viterbi algorithm with respect to symbol length, investigations were limited to a symbol size of five by compute resources. Provided with more compute resources, or a more efficient version of the decoding algorithm, work could be carried out which further explores the relationship between error correction and symbol length.

Bibliography

[1] David Reinsel-John Gantz-John Rydning, J Reinsel, and J Gantz. Data age 2025: The digitization of the world from edge to core. *International Data Corporation & Seagate*, 16, 2018. URL https://www.seagate.com/files/www-content/our-story/trends/files/idc-seagate-dataage-whitepaper.pdf.

[2] Richard P. Feynman. There's plenty of room at the bottom [data storage]. *Journal of Microelectromechanical Systems*, 1(1):60–66, 1992. doi: 10.1109/84.128057. URL https://ieeexplore.ieee.org/document/128057.

[3] George M. Church, Yuan Gao, and Sriram Kosuri. Next-generation digital information storage in dna. *Science*, 337(6102):1628–1628, 2012. doi: 10.1126/science.1226355. URL https://www.science.org/doi/abs/10.1126/science.1226355.

[4] Victor Zhirnov, Reza M. Zadegan, Gurtej S. Sandhu, George M. Church, and William L. Hughes. Nucleic acid memory. *Nature Materials*, 15(4):366–370, Apr 2016. ISSN 1476-4660. doi: 10.1038/nmat4594. URL https://doi.org/10.1038/nmat4594.

[5] Martin G. T. A. Rutten, Frits W. Vaandrager, Johannes A. A. W. Elemans, and Roeland J. M. Nolte. Encoding information into polymers. *Nature Reviews Chemistry*, 2(11):365–381, Nov 2018. ISSN 2397-3358. doi: 10.1038/s41570-018-0051-5. URL https://doi.org/10.1038/s41570-018-0051-5.

[6] Luis Ceze, Jeff Nivala, and Karin Strauss. Molecular digital data storage using dna. *Nature Reviews Genetics*, 20(8):456–466, Aug 2019. ISSN 1471-0064. doi: 10.1038/s41576-019-0125-3. URL https://doi.org/10.1038/s41576-019-0125-3.

[7] Christopher N. Takahashi, Bichlien H. Nguyen, Karin Strauss, and Luis Ceze. Demonstration of end-to-end automation of dna data storage. *Scientific Reports*, 9(1):4998, Mar 2019. ISSN 2045-2322. doi: 10.1038/s41598-019-41228-8. URL https://doi.org/10.1038/s41598-019-41228-8.

[8] Morten E. Allentoft, Matthew Collins, David Harker, James Haile, Charlotte L. Oskam, Marie L. Hale, Paula F. Campos, Jose A. Samaniego, Thomas P.M. Gilbert, Eske Willerslev, Guojie Zhang, R. Paul Scofield, Richard N. Holdaway, and Michael Bunce. The half-life of dna in bone: measuring decay kinetics in 158 dated fossils. *Proceedings of the Royal Society B: Biological Sciences*, 2012. doi: 10.1098/rspb.2012.1745. URL http://doi.org/10.1098/rspb.2012.1745.

[9] Ludovic Orlando, Aurélien Ginolhac, Guojie Zhang, Duane Froese, Anders Albrechtsen, Mathias Stiller, Mikkel Schubert, Enrico Cappellini, Bent Petersen, Ida Moltke, Philip L. F. Johnson, Matteo Fumagalli, Julia T. Vilstrup, Maanasa Raghavan, Thorfinn Korneliussen, Anna-Sapfo Malaspinas, Josef Vogt, Damian Szklarczyk, Christian D. Kelstrup, Jakob Vinther, Andrei Dolocan, Jesper Stenderup, Amhed M. V. Velazquez, James Cahill, Morten Rasmussen, Xiaoli Wang, Jiumeng Min, Grant D. Zazula, Andaine Seguin-Orlando, Cecilie Mortensen, Kim Magnussen, John F. Thompson, Jacobo Weinstock, Kristian Gregersen, Knut H. Røed, Véra Eisenmann, Carl J. Rubin, Donald C. Miller, Douglas F. Antczak, Mads F. Bertelsen, Søren Brunak, Khaled A. S. Al-Rasheid, Oliver Ryder, Leif Andersson, John Mundy, Anders Krogh, M. Thomas P. Gilbert, Kurt Kjær, Thomas Sicheritz-Ponten, Lars Juhl Jensen, Jesper V. Olsen, Michael Hofreiter, Rasmus Nielsen, Beth Shapiro, Jun Wang, and Eske Willerslev. Recalibrating equus evolution using the genome sequence of an early middle pleistocene horse. *Nature*, 499(7456):74–78, Jul 2013. ISSN 1476-4687. doi: 10.1038/nature12323. URL https://doi.org/10.1038/nature12323.

[10] Andy Extance. How dna could store all the world's data. *Nature*, 537(7618):22–24, Sep 2016. ISSN 1476-4687. doi: 10.1038/537022a. URL https://doi.org/10.1038/537022a.

[11] 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, Feb 2013. ISSN 1476-4687. doi: 10.1038/nature11875. URL https://doi.org/10.1038/nature11875.

[12] The Global Crop Diversity Trust. Svalbard global seed vault, 2023. URL https://www.croptrust.org/work/svalbard-global-seed-vault/. (accessed 21 Jan 2023).

[13] Omer S. Sella, Amir Apelbaum, Thomas Heinis, Jasmine Quah, and Andrew W. Moore. Dna archival storage, a bottom up approach. In *Proceedings of the 13th ACM Workshop on Hot Topics in Storage and File Systems*, HotStorage '21, page 58–63, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450385503. doi: 10.1145/3465332.3470880. URL https://doi.org/10.1145/3465332.3470880.

[14] D.P. Snustad and M.J. Simmons. *Principles of Genetics*. Wiley, 2015. ISBN 9781119142287. URL https://www.wiley.com/en-us/Principles+of+Genetics,+7th+Edition-p-9781119142287.

[15] Gregor Mendel. Versuche Über pflanzen-hybriden. *Verhandlungen des naturforschenden Vereines in Brunn*, 4, 01 1866. doi: 10.1093/oxfordjournals.jhered.a106151. URL https://doi.org/10.1093/oxfordjournals.jhered.a106151.

[16] Oswald T. Avery, Colin M. MacLeod, and Maclyn McCarty. Studies of the chemical nature of the substance inducing transformation of pneumococcal types. induction of transformation by a deoxyribonucleic acid fraction isolated from pneumococcus type iii. *Journal of Experimental Medicine*, 79(2):137–158, 02 1944. ISSN 0022-1007. doi: 10.1084/jem.79.2.137. URL https://doi.org/10.1084/jem.79.2.137.

[17] W.S. Klug, M.R. Cummings, S.M. Ward, and C. Spencer. *Concepts of Genetics*. Pearson International ed. Pearson Benjamin Cummings, 2009. ISBN 9780321524041. URL https://books.google.co.uk/books?id=wD7PQgAACAAJ.

[18] J. D. Watson and F. H. C. Crick. Molecular structure of nucleic acids: A structure for deoxyribose nucleic acid. *Nature*, 171(4356):737–738, Apr 1953. ISSN 1476-4687. doi: 10.1038/171737a0. URL https://doi.org/10.1038/171737a0.

[19] M. H. F. Wilkins, A. R. Stokes, and H. R. Wilson. Molecular structure of nucleic acids: Molecular structure of deoxypentose nucleic acids. *Nature*, 171(4356):738–740, Apr 1953. ISSN 1476-4687. doi: 10.1038/171738a0. URL https://doi.org/10.1038/171738a0.

[20] Rosalind E. Franklin and R. G. Gosling. Molecular configuration in sodium thymonucleate. *Nature*, 171(4356):740–741, Apr 1953. ISSN 1476-4687. doi: 10.1038/171740a0. URL https://doi.org/10.1038/171740a0.

[21] Leonard M. Adleman. Molecular computation of solutions to combinatorial problems. *Science*, 266(5187):1021–1024, 1994. doi: 10.1126/science.7973651. URL https://www.science.org/doi/abs/10.1126/science.7973651.

[22] Richard J. Lipton. Dna solution of hard computational problems. *Science*, 268(5210):542–545, 1995. doi: 10.1126/science.7725098. URL https://www.science.org/doi/abs/10.1126/science.7725098.

[23] Norbert Wiener. Machines smarter than men? interview with dr. norbert wiener, noted scientist. *U.S. News World Report*, 56:84–87, 1964. URL https://profiles.nlm.nih.gov/101584906X7699.

[24] MS Neiman. On the molecular memory systems and the directed mutations. *Radiotekhnika*, 6:1–8, 1965. URL https://sites.google.com/site/msneiman1905/eng. (in Russian).

[25] Eric B. Baum. Building an associative memory vastly larger than the brain. *Science*, 268(5210):583–585, 1995. doi: 10.1126/science.7725109. URL https://www.science.org/doi/abs/10.1126/science.7725109.

[26] Catherine Taylor Clelland, Viviana Risca, and Carter Bancroft. Hiding messages in dna microdots. *Nature*, 399(6736):533–534, Jun 1999. ISSN 1476-4687. doi: 10.1038/21092. URL https://doi.org/10.1038/21092.

[27] Shubham Chandak, Joachim Neu, Kedar Tatwawadi, Jay Mardia, Billy Lau, Matthew Kubit, Reyna Hulett, Peter Griffin, Mary Wootters, Tsachy Weissman, and Hanlee Ji. Overcoming high nanopore basecaller error rates for dna storage via basecaller-decoder integration and convolutional codes. In *ICASSP 2020 - 2020 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 8822–8826, 2020. doi: 10.1109/ICASSP40776.2020.9053441. URL https://ieeexplore.ieee.org/document/9053441.

[28] Ryan R. Wick, Louise M. Judd, and Kathryn E. Holt. Performance of neural network basecalling tools for oxford nanopore sequencing. *Genome Biology*, 20(1):129, Jun 2019. ISSN 1474-760X. doi: 10.1186/s13059-019-1727-y. URL https://doi.org/10.1186/s13059-019-1727-y.

[29] Kevin N. Lin, Kevin Volkel, James M. Tuck, and Albert J. Keung. Dynamic and scalable dna-based information storage. *Nature Communications*, 11(1):2981, Jun 2020. ISSN 2041-1723. doi: 10.1038/s41467-020-16797-2. URL https://doi.org/10.1038/s41467-020-16797-2.

[30] Billy Lau, Shubham Chandak, Sharmili Roy, Kedar Tatwawadi, Mary Wootters, Tsachy Weissman, and Hanlee P. Ji. Magnetic dna random access memory with nanopore readouts and exponentially-scaled combinatorial addressing. *bioRxiv*, 2021. doi: 10.1101/2021.09.15.460571. URL https://www.biorxiv.org/content/early/2021/09/16/2021.09.15.460571.

[31] 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(1):14138, Sep 2015. ISSN 2045-2322. doi: 10.1038/srep14138. URL https://doi.org/10.1038/srep14138.

[32] James Bornholt, Randolph Lopez, Douglas M. Carmean, Luis Ceze, Georg Seelig, and Karin Strauss. A dna-based archival storage system. *SIGARCH Comput. Archit. News*, 44(2):637–649, mar 2016. ISSN 0163-5964. doi: 10.1145/2980024.2872397. URL https://doi.org/10.1145/2980024.2872397.

[33] Lee Organick, Siena Dumas Ang, Yuan-Jyue Chen, Randolph Lopez, Sergey Yekhanin, Konstantin Makarychev, Miklos Z. Racz, Govinda Kamath, Parikshit Gopalan, Bichlien Nguyen, Christopher N. 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. Random access in large-scale dna data storage. *Nature Biotechnology*, 36(3):242–248, Mar 2018. ISSN 1546-1696. doi: 10.1038/nbt.4079. URL https://doi.org/10.1038/nbt.4079.

[34] Thomas Heinis and Jamie J. Alnasir. Survey of information encoding techniques for dna, 2019. URL https://arxiv.org/abs/1906.11062.

[35] Philipp L. Antkowiak, Jory Lietard, Mohammad Zalbagi Darestani, Mark M. Somoza, Wendelin J. Stark, Reinhard Heckel, and Robert N. Grass. Low cost dna data storage using photolithographic synthesis and advanced information reconstruction and error correction. *Nature Communications*, 11(1):5345, Oct 2020. ISSN 2041-1723. doi: 10.1038/s41467-020-19148-3. URL https://doi.org/10.1038/s41467-020-19148-3.

[36] Ravi U. Sheth and Harris H. Wang. Dna-based memory devices for recording cellular events. *Nature Reviews Genetics*, 19(11):718–732, Nov 2018. ISSN 1471-0064. doi: 10.1038/s41576-018-0052-8. URL https://doi.org/10.1038/s41576-018-0052-8.

[37] Federico Tavella, Alberto Giaretta, Triona Marie Dooley-Cullinane, Mauro Conti, Lee Coffey, and Sasitharan Balasubramaniam. Dna molecular storage system: Transferring digitally encoded information through bacterial nanonetworks. *IEEE Transactions on Emerging Topics in Computing*, 9(3):1566–1580, 2021. doi: 10.1109/TETC.2019.2932685.

[38] James L. Banal, Tyson R. Shepherd, Joseph Berleant, Hellen Huang, Miguel Reyes, Cheri M. Ackerman, Paul C. Blainey, and Mark Bathe. Random access dna memory using boolean search in an archival file storage system. *Nature Materials*, 20(9):1272–1280, Sep 2021. ISSN 1476-4660. doi: 10.1038/s41563-021-01021-3. URL https://doi.org/10.1038/s41563-021-01021-3.

[39] Luca Piantanida and William L. Hughes. A pcr-free approach to random access in dna. *Nature Materials*, 20(9):1173–1174, Sep 2021. ISSN 1476-4660. doi: 10.1038/s41563-021-01089-x. URL https://doi.org/10.1038/s41563-021-01089-x.

[40] C. E. Shannon. A mathematical theory of communication. *The Bell System Technical Journal*, 27(3):379–423, 1948. doi: 10.1002/j.1538-7305.1948.tb01338.x.

[41] R. W. Hamming. Error detecting and error correcting codes. *The Bell System Technical Journal*, 29(2):147–160, 1950. doi: 10.1002/j.1538-7305.1950.tb00463.x.

[42] Reinhard Heckel, Ilan Shomorony, Kannan Ramchandran, and David N. C. Tse. Fundamental limits of dna storage systems. In *2017 IEEE International Symposium on Information Theory (ISIT)*, pages 3130–3134, 2017. doi: 10.1109/ISIT.2017.8007106. URL https://ieeexplore.ieee.org/document/8007106.

[43] Reinhard Heckel, Gediminas Mikutis, and Robert N. Grass. A characterization of the dna data storage channel. *Scientific Reports*, 9(1):9663, Jul 2019. ISSN 2045-2322. doi: 10.1038/s41598-019-45832-6. URL https://doi.org/10.1038/s41598-019-45832-6.

[44] Omer Sella. *Coding for emerging archival storage media*. PhD thesis, University of Cambridge, 2022.

[45] Xiaotu Ma, Ying Shao, Liqing Tian, Diane A. Flasch, Heather L. Mulder, Michael N. Edmonson, Yu Liu, Xiang Chen, Scott Newman, Joy Nakitandwe, Yongjin Li, Benshang Li, Shuhong Shen, Zhaoming Wang, Sheila Shurtleff, Leslie L. Robison, Shawn Levy, John Easton, and Jinghui Zhang. Analysis of error profiles in deep next-generation sequencing data. *Genome Biology*, 20(1):50, Mar 2019. ISSN 1474-760X. doi: 10.1186/s13059-019-1659-6. URL https://doi.org/10.1186/s13059-019-1659-6.

[46] Hugues Mercier and Vijay K. Bhargava. Convolutional codes for channels with deletion errors. In *2009 11th Canadian Workshop on Information Theory*, pages 136–139, 2009. doi: 10.1109/CWIT.2009.5069539.

Glossary

amplification Creating more copies of a sequence of DNA, usually through a polymerase chain reaction. 4

basecaller A program which maps a signal from a DNA sequencer to a sequence of nucleotides. 14

degenerate bases Two different bases that occupy the same base position on different instances of a particular oligonucleotide. 12

energy barrier A model in which information is modelled as a resource protected by an 'energy barrier', which is a barrier that requires a certain amount of energy to break through. If the information-carrying particles inside the barrier have enough energy, they can 'jump above' the barrier, leading to information loss (this would imply, for example, that higher temperatures would lead to greater information loss). Information loss by particles gaining enough energy to 'jump' the barrier can be thought of as 'classical' information loss. It is also possible to lose information due to quantum effects if the information-bearing particles are small enough and very densely packed (e.g. very densely packed electrons may 'tunnel' through an energy barrier, leading to quantum information loss). 16

GC content The ratio between the count of G and C instances in a sequence of DNA, divided by the length of the sequence. 7

GC limits Upper and lower bounds on the GC content. 20

GC variation Variation in GC content between equal length subsequences of DNA. 6

homopolymer Subsequences that are made up of repetitions of a single base, a special case of a short tandem repeat. 5

information density In general, information density refers to the number of bits per unit volume in storage media. In the case of DNA, our units for volume and mass are set - the minimum unit of storage is a nucleotide, so information density is measured in bits per nucleotide. 7

logical redundancy This refers to an error-correcting code that stores information in a way which allows the detection and correction of a certain number of errors per sequence of nucleotides. 12

oligonucleotide A short DNA or RNA strand, often synthesised by a machine and used for genetic testing/research or DNA storage/computation. 10

physical redundancy This refers to having many copies of the same sequence of nucleotides. Since errors present in one copy may not be present in another, having large physical redundancy decreases the error rate (unless the errors are systematic). 12

primer A short nucleic acid sequence which provides the starting point for a polymerase chain reaction. 5

secondary structure The structure of a DNA molecule in terms of the bonding of base pairs. It may describe the bonding of two complementary strands of DNA or the self-bonding within a single molecule where different parts of the base sequence happen to be complementary to each other. 12

Terminal Deoxynucleotidyl Transferases (TdTs) Specialised DNA polymerase enzymes which can be used in DNA synthesis. 13

Viterbi algorithm An algorithm used to decode convolutional codes. Expanded upon in section 4.2.5. 15

Appendix A

Algorithm Implementations

A.1 FSM Generation

Below is the Rust code used to construct the finite state machine. First, the data FSM and transition table data structures are defined.

```
type FSM = (usize, usize, String, Table);
type Table = HashMap<String, HashMap<String, (String, String)>>;
```

Next, the function for generating an FSM is defined.

```
fn construct_fsm(
    input_size: usize,
    output_size: usize,
    init_state: String,
    constraints: Constraints,
    mechanism: impl FnMut(&str, &str, Vec<String>) -> String,
) -> FSM {
    return (
        input_size,
        output_size,
        init_state,
        construct_table(input_size, output_size, constraints, mechanism),
    );
}
```

Lastly, the helper functions for transition table construction are defined.

```
fn construct_table(
    input_size: usize, output_size: usize, constraints: Constraints,
    mut mechanism: impl FnMut(&str, &str, Vec<String>) -> String,
) -> Table {
    assert!(input_size < output_size);

    let mut table: Table = HashMap::new();

    let reserved_size = output_size - input_size;
    let states = populate_space(output_size);
    let inputs = populate_space(input_size);
    let reserved = populate_space(reserved_size);

    for s in &states {
        table.insert(s.clone(), HashMap::new());
        for i in &inputs {
            let mut candidates: Vec<String> = Vec::new();
            for r in &reserved {
                if constraints.satisfied(bits_to_dna(&(s.to_owned() + r + i))) {
                    candidates.push(r.to_string());
                }
            }
            if candidates.is_empty() {
                panic!("Impossible to meet constraints!");
            }

            let chosen = mechanism(&s, &i, candidates);
            let output = chosen + &i;
            table
                .get_mut(s)
                .unwrap()
                .insert(i.to_string(), (output.clone(), output));
        }
    }

    return table;
}
```

```rust

fn populate_space(size: usize) -> Vec<String> {
    let mut space: Vec<String> = Vec::new();
    let dim = usize::pow(2, size as u32);
    for i in 0..dim {
        let bin = format!("{i:b}");
        let padding = size - bin.len();
        space.push("0".repeat(padding) + &bin);
    }
    return space;
}
```

A.2 Convolutional Encoding and Viterbi Decoding

Below is the Rust code used to carry out the convolutional coding and Viterbi decoding.

```rust
use std::collections::HashMap;
type Table = HashMap<String, HashMap<String, (String, String)>>;

struct FSM {
    input_size: usize,
    output_size: usize,
    init_state: String,
    table: Table,
}

struct Path {
    length: usize,
    sequence: String,
    observations: String,
}

impl Path {
    fn extend(&mut self, dist: usize, input: String, output: String) {
        self.length += dist;
        self.sequence += &input;
        self.observations += &output;
    }
}

impl Clone for Path {
    fn clone(&self) -> Self {
        Path {
            length: self.length,
            sequence: self.sequence.clone(),
            observations: self.observations.clone(),
        }
    }
}

type Paths = HashMap<String, Path>;

fn hamming_dist(one: &str, two: &str) -> usize {
    assert!(one.len() == two.len());
    return one.chars().zip(two.chars()).filter(|(a, b)| a != b).count();
}

fn conv(fsm: &FSM, msg: String) -> String {
    assert!(msg.len() % fsm.input_size == 0);

    let mut state = &fsm.init_state;
    let mut result = String::from("");

    for i in (0..msg.len()).step_by(fsm.input_size) {
        let input = &msg[i..i + fsm.input_size];
        let (next, output) = &fsm.table[state][input];
        result += output;
        state = next;
    }

    return result;
}

fn viterbi(fsm: &FSM, msg: String) -> Path {
    assert!(msg.len() % fsm.output_size == 0);

    let start_path = Path {
        length: 0,
        sequence: String::from(""),
```

```
        observations: String::from(""),
    };
    let mut paths: Paths = HashMap::from([(fsm.init_state.to_string(), start_path)]);

    for i in (0..msg.len()).step_by(fsm.output_size) {
        let symbol = &msg[i..i + fsm.output_size];
        let mut extended_paths: Paths = HashMap::new();

        for (tip, path) in paths {
            for (input, (next, output)) in &fsm.table[&tip] {
                let dist = hamming_dist(symbol, output);
                let mut extended = path.clone();
                extended.extend(dist, input.to_string(), output.to_string());

                if !extended_paths.contains_key(next)
                    || extended.length < extended_paths[next].length
                {
                    extended_paths.insert(next.to_string(), extended);
                }
            }
        }

        paths = extended_paths;
    }

    return paths
        .values()
        .reduce(|a, b| if a.length < b.length { a } else { b })
        .unwrap()
        .clone();
}
```

A.3 Error Injection

Many different methods for error injection were used throughout this project for various purposes. The code used to perform these injections is presented below.

A.3.1 Random Error Injection

```
def rand_base(exclude: str = ""):
    bases = "ACGT".replace(exclude, "")
    return rn.choice(bases)


def inject_base_errors(message: str, rate: float = 0.01) -> str:
    result = ""
    for base in message:
        if rn.random() <= rate:
            result += rand_base(base)
        else:
            result += base
    return result
```

A.3.2 Exact Error Injection

```
def rand_base(exclude: str = ""):
    bases = "ACGT".replace(exclude, "")
    return rn.choice(bases)


# Injects a precise number of errors at random locations.
def inject_base_errors_exact(message: str, num: int) -> str:
    result = list(message)
    if len(message) < num:
        raise Exception("Number of errors too high for message length.")
    locations = rn.sample(range(len(message)), num)
    for i in locations:
        result[i] = rand_base(exclude=result[i])
    return "".join(result)
```

A.3.3 Burst Error Injection

```
def inject_burst_errors(
    message: str, rate: float = 0.01, min_burst=2, max_burst=4
) -> str:
    result = message[:]

```

```
    num_errors = int(len(message) * rate)

    # Injects ROUGHLY num_errors many errors into the message.
    while num_errors > 0:
        burst = rn.randint(min_burst, max_burst)
        position = rn.randrange(len(message) - burst)

        error = ""
        for i in range(burst):
            error += rand_base(message[position + i])

        result = result[:position] + error + result[position + burst :]

        num_errors -= burst

    return result

def rand_base(exclude: str = ""):
    bases = "ACGT".replace(exclude, "")
    return rn.choice(bases)
```

A.3.4 Insertions and Deletions

```
def rand_base(exclude: str = ""):
    bases = "ACGT".replace(exclude, "")
    return rn.choice(bases)


def inject_deletion_errors(message: str, rate: float = 0.01) -> str:
    result = ""
    for base in message:
        if rn.random() > rate:
            result += base
    return result


def inject_insertion_errors(message: str, rate: float = 0.01) -> str:
    result = ""
    for base in message:
        if rn.random() <= rate:
            result += rand_base()
        result += base
    return result
```

A.4 Choice Mechanisms

A.4.1 GC-Tracking

```
fn gc_tracking(state: &str, input: &str, reserved: Vec<String>) -> String {
    let mut closest: String = reserved[0].clone();
    let mut min_diff: f32 = 1.0;
    let gc_target: f32 = 0.5;

    for candidate in reserved {
        let concat = state.to_string() + &candidate + input;
        let gc_content = gc_content(&bits_to_dna(&concat));
        let diff = (gc_target - gc_content).abs();

        if diff == 0.0 {
            return candidate;
        }

        if diff < min_diff {
            min_diff = diff;
            closest = candidate;
        }
    }

    return closest;
}
```

A.4.2 Randomised GC-Tracking

```
fn gc_tracked_random(
    state: &str,
    input: &str,
    reserved: Vec<String>,
    rng: &mut StdRng,
) -> String {
```

```
    let mut closest: Vec<String> = Vec::new();
    let mut min_diff: f32 = 1.0;
    let gc_target: f32 = 0.5;

    for candidate in reserved {
        let concat = state.to_string() + &candidate + input;
        let gc_content = gc_content(&bits_to_dna(&concat));
        let diff = (gc_target - gc_content).abs();

        if diff < min_diff {
            min_diff = diff;
            closest.clear();
            closest.push(candidate);
        } else if diff == min_diff {
            closest.push(candidate);
        }
    }

    return closest.choose(rng).unwrap().to_string();
}
```

A.4.3 Most Similar

```
fn most_similar(
    state: &str,
    input: &str,
    reserved: Vec<String>,
    rng: &mut StdRng,
) -> String {
    assert!(input.len() * 2 == state.len());

    let mut closest: Vec<String> = Vec::new();
    let mut min_diff: usize = input.len();

    for candidate in reserved {
        let diff = hamming_dist(input, &candidate);

        if diff < min_diff {
            min_diff = diff;
            closest.clear();
            closest.push(candidate);
        } else if diff == min_diff {
            closest.push(candidate);
        }
    }

    return closest.choose(rng).unwrap().to_string();
}
```

A.4.4 Most Different

```
fn most_different(
    state: &str,
    input: &str,
    reserved: Vec<String>,
    rng: &mut StdRng,
) -> String {
    assert!(input.len() * 2 == state.len());

    let mut closest: Vec<String> = Vec::new();
    let mut max_diff: usize = 0;

    for candidate in reserved {
        let diff = hamming_dist(input, &candidate);

        if diff > max_diff {
            max_diff = diff;
            closest.clear();
            closest.push(candidate);
        } else if diff == max_diff {
            closest.push(candidate);
        }
    }

    return closest.choose(rng).unwrap().to_string();
}
```

A.4.5 Random Unused

```
static mut USED: Lazy<HashMap<String, Vec<String>>> = Lazy::new(|| HashMap::new());
```

```
fn random_unused(input: &str, reserved: Vec<String>, rng: &mut StdRng) -> String {
    let mut diff: Vec<String> = vec![];

    unsafe {
        if !USED.contains_key(input) {
            USED.insert(input.to_string(), Vec::new());
        }

        let used = USED.get(input).unwrap();
        for r in &reserved {
            if !used.contains(r) {
                diff.push(r.to_string());
            }
        }

        if diff.is_empty() {
            return reserved.choose(rng).unwrap().to_string();
        }

        let choice = diff.choose(rng).unwrap().to_string();
        USED.get_mut(input).unwrap().push(choice.clone());
        return choice;
    }
}
```

A.4.6 Alternating Parity

```
fn even_parity(seq: &str) -> bool {
    return seq.chars().filter(|c| *c == '1').count() % 2 == 0;
}

fn alt_parity(
    state: &str,
    input: &str,
    reserved: Vec<String>,
    rng: &mut StdRng,
) -> String {
    let mut alt: Vec<String> = Vec::new();
    let state_parity = even_parity(state);
    for candidate in &reserved {
        let next_parity = even_parity(&(candidate.to_owned() + input));
        if state_parity != next_parity {
            alt.push(candidate.to_string());
        }
    }

    if alt.is_empty() {
        return random_choice(reserved, rng);
    }

    return alt.choose(rng).unwrap().to_string();
}
```

The alternating parity choice mechanism also requires a corresponding change to the decoding algorithm to take advantage of the parity information. The new updated algorithm is shown below.

```
fn even_parity(seq: &str) -> bool {
    return seq.chars().filter(|c| *c == '1').count() % 2 == 0;
}

fn viterbi(fsm: &FSM, msg: String) -> Path {
    assert!(msg.len() % fsm.output_size == 0);

    let start_path = Path {
        length: 0,
        sequence: String::from(""),
        observations: String::from(""),
    };
    let mut paths: Paths = HashMap::from([(fsm.init_state.to_string(), start_path)]);

    for i in (0..msg.len()).step_by(fsm.output_size) {
        let symbol = &msg[i..i + fsm.output_size];
        let mut extended_paths: Paths = HashMap::new();

        for (tip, path) in paths {
            for (input, (next, output)) in &fsm.table[&tip] {
                let dist = hamming_dist(symbol, output);
                let mut extended = path.clone();
                extended.extend(dist, input.to_string(), output.to_string());

```

```
                if !extended_paths.contains_key(next)
                    || extended.length < extended_paths[next].length
                {
                    extended_paths.insert(next.to_string(), extended);
                } else if extended.length == extended_paths[next].length {
                    let len = extended.observations.len();
                    if len > fsm.output_size * 2 {
                        let prev = &extended.observations
                            [len - fsm.output_size * 2..len - fsm.output_size];
                        let curr = &extended.observations[len - fsm.output_size..len];
                        let path_prev = &extended_paths[next].observations
                            [len - fsm.output_size * 2..len - fsm.output_size];
                        let path_curr =
                            &extended_paths[next].observations[len - fsm.output_size..len
    ];

                        if even_parity(path_prev) == even_parity(path_curr)
                            && even_parity(prev) != even_parity(curr)
                        {
                            extended_paths.insert(next.to_string(), extended);
                        }
                    }
                }
            }
        }

        paths = extended_paths;
    }

    return paths
        .values()
        .reduce(|a, b| if a.length < b.length { a } else { b })
        .unwrap()
        .clone();
}
```

A.5 Constraint Decoding

Below is the code used to integrate the constraint list into the decoding algorithm.

```
pub(crate) fn constraint_viterbi(fsm: &FSM, constraints: Constraints, msg: String) ->
    Path {
    assert!(msg.len() % fsm.output_size == 0);

    let start_path = Path {
        length: 0,
        sequence: String::from(""),
        observations: String::from(""),
    };
    let mut paths: Paths = HashMap::from([(fsm.init_state.to_string(), start_path)]);

    for i in (0..msg.len()).step_by(fsm.output_size) {
        let symbol = &msg[i..i + fsm.output_size];
        let mut extended_paths: Paths = HashMap::new();

        for (tip, path) in paths {
            for (input, (next, output)) in &fsm.table[&tip] {
                let dist = hamming_dist(symbol, output);
                let mut extended = path.clone();
                extended.extend(dist, input.to_string(), output.to_string());

                // Add Constraint Penalty
                let len = extended.observations.len();
                if len > fsm.output_size * 2 {
                    if !constraints.satisfied(
                        extended.observations[len - fsm.output_size * 2..len].to_string()
    ,
                    ) {
                        extended.length += 1000;
                    }
                }

                if !extended_paths.contains_key(next)
                    || extended.length < extended_paths[next].length
                {
                    extended_paths.insert(next.to_string(), extended);
                }
            }
        }

        paths = extended_paths;
```

```
    }

    return paths
        .values()
        .reduce(|a, b| if a.length < b.length { a } else { b })
        .unwrap()
        .clone();
}
```

A.6 Constraints and STRs

A.6.1 Original Constraints Implementation

```
struct Constraints {
    gc_min: f32,
    gc_max: f32,
    max_run_length: usize,
    reserved: Vec<String>,
}

impl Constraints {
    pub(crate) fn satisfied(&self, seq: String) -> bool {
        let gc_content: f32 = gc_content(&seq);

        return self.gc_min <= gc_content
            && gc_content <= self.gc_max
            && longest_homopolymer(&seq) <= self.max_run_length
            && !contains_subsequence(&self.reserved, &seq);
    }
}

fn gc_content(seq: &str) -> f32 {
    return seq.chars().filter(|c| *c == 'G' || *c == 'C').count() as f32 / seq.len() as
    f32;
}

fn longest_homopolymer(seq: &str) -> usize {
    let mut longest: usize = 0;
    let mut count: usize = 0;
    let mut curr = seq.chars().nth(0).unwrap();

    for base in seq.chars() {
        if base == curr {
            count += 1;
            if count > longest {
                longest = count;
            }
        } else {
            count = 1;
        }

        curr = base;
    }

    return longest;
}

fn contains_subsequence(reserved: &Vec<String>, seq: &str) -> bool {
    return reserved.into_iter().any(|r| seq.contains(r));
}
```

A.6.2 STR Checking Constraints Implementation

```
struct Constraints {
    gc_min: f32,
    gc_max: f32,
    str_lower: usize,
    str_upper: usize,
    max_run_length: usize,
    reserved: Vec<String>,
}

impl Constraints {
    fn satisfied(&self, seq: String) -> bool {
        assert!(self.str_upper * 2 <= seq.len());
        assert!(self.str_lower <= self.str_upper);

        let gc_content: f32 = gc_content(&seq);
```

```
16
17          return self.gc_min <= gc_content
18              && gc_content <= self.gc_max
19              && longest_homopolymer(&seq) <= self.max_run_length
20              && !str_present(&seq, self.str_lower, self.str_upper)
21              && !contains_subsequence(&self.reserved, &seq);
22      }
23  }
24
25  fn gc_content(seq: &str) -> f32 {
26      return seq.chars().filter(|c| *c == 'G' || *c == 'C').count() as f32 / seq.len() as
        f32;
27  }
28
29  fn str_present(seq: &str, lower: usize, upper: usize) -> bool {
30      for size in lower..=upper {
31          for i in 0..=seq.len() - (size * 2) {
32              if seq[i..i + size] == seq[i + size..i + (size * 2)] {
33                  return true;
34              }
35          }
36      }
37      return false;
38  }
39
40  fn longest_homopolymer(seq: &str) -> usize {
41      let mut longest: usize = 0;
42      let mut count: usize = 0;
43      let mut curr = seq.chars().nth(0).unwrap();
44
45      for base in seq.chars() {
46          if base == curr {
47              count += 1;
48              if count > longest {
49                  longest = count;
50              }
51          } else {
52              count = 1;
53          }
54
55          curr = base;
56      }
57
58      return longest;
59  }
60
61  fn contains_subsequence(reserved: &Vec<String>, seq: &str) -> bool {
62      return reserved.into_iter().any(|r| seq.contains(r));
63  }
```

A.6.3 STR Check Function

```
1  fn str_present(seq: &str, lower: usize, upper: usize) -> bool {
2      for size in lower..=upper {
3          for i in 0..=seq.len() - (size * 2) {
4              if seq[i..i + size] == seq[i + size..i + (size * 2)] {
5                  return true;
6              }
7          }
8      }
9      return false;
10 }
```

A.6.4 Counting STRs

This function is used to count STRs for the results in section 6.3. It counts each repetition of an STR separately. Therefore the string "ACGACGACGACG" would result in a length-three STR count of four.

```
1  def strs(sequence: str, size: int = 4) -> int:
2      count = 0
3      for i in range(len(sequence) - size * 2):
4          one = sequence[i : i + size]
5          two = sequence[i + size : i + size * 2]
6          if one == two:
7              count += 1
8      return count
```

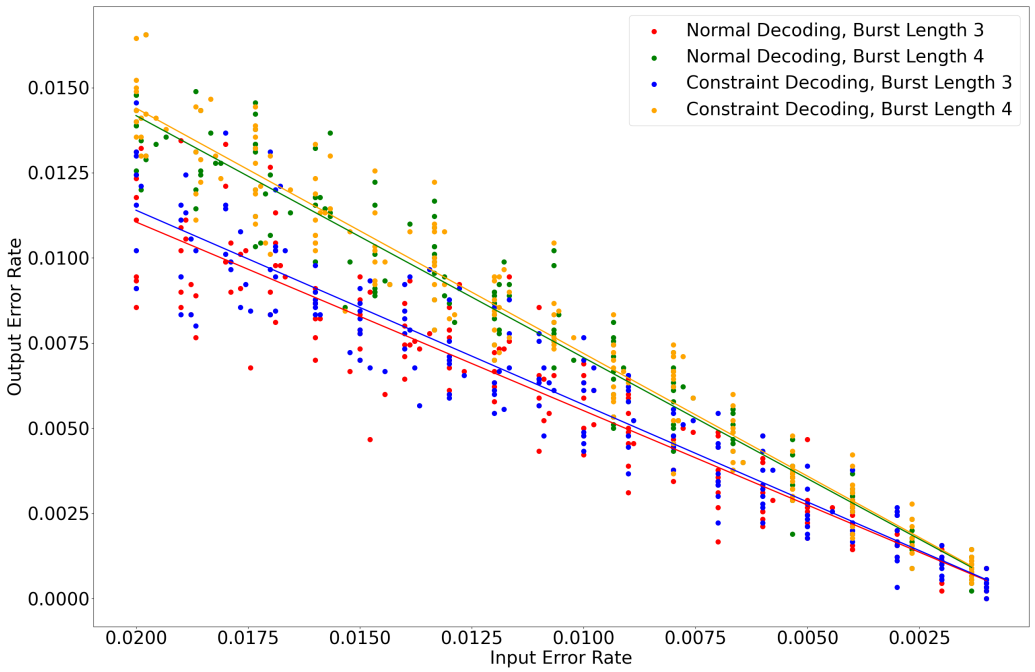


Figure A.1: The effect of constraint-based decoding on burst errors. No improvement was observed when using constraint decoding compared to normal Viterbi decoding. The symbol size and number of reserved bit are both four.

Appendix B

Profiling Results

The full results of the Python and Rust profiling. Interestingly convolutional coding takes longer with a call to the Rust library than through native Python. The explanation for this is almost certainly that convolutional coding in Python is fast enough that the overhead of calling into a compiled Rust library negates any performance gain from using Rust. However, as can be seen in the figure, as the sequence lengths increase, the time taken to perform the encoding increases. This means that at some longer sequence length, the Rust call will almost certainly become the faster option.

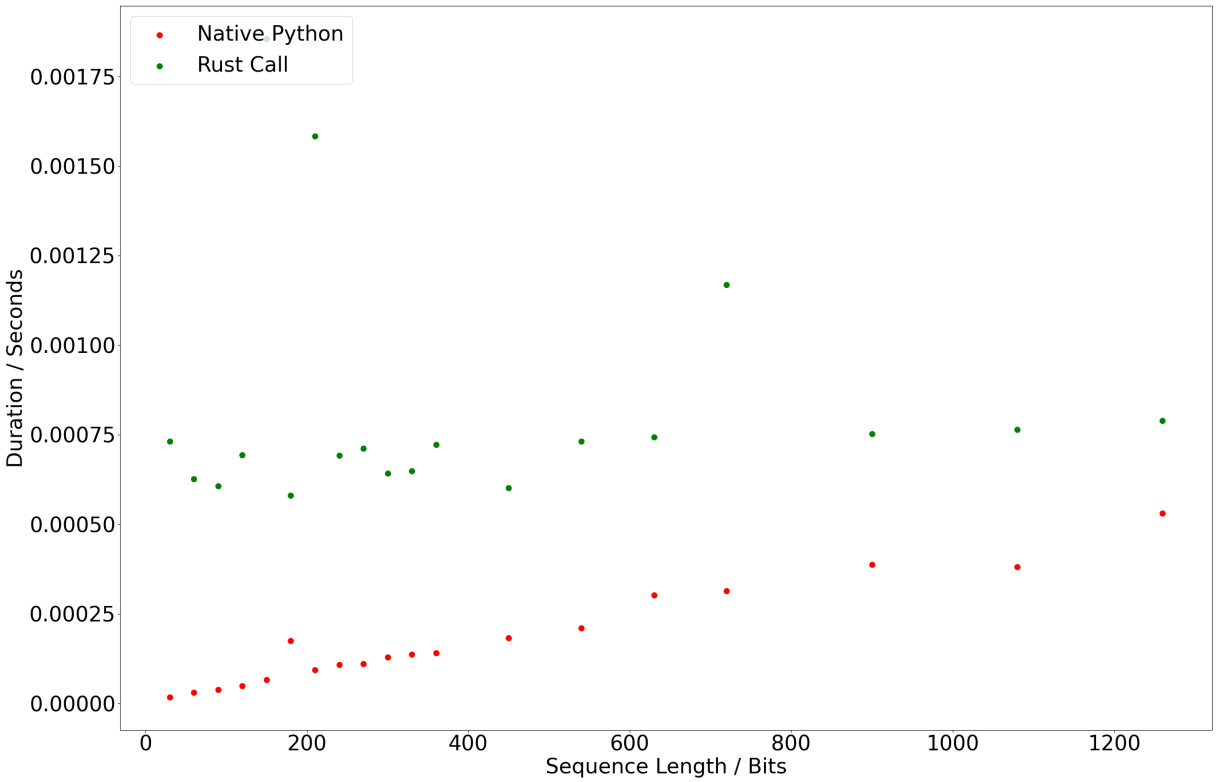


Figure B.1: Convolutional Coding

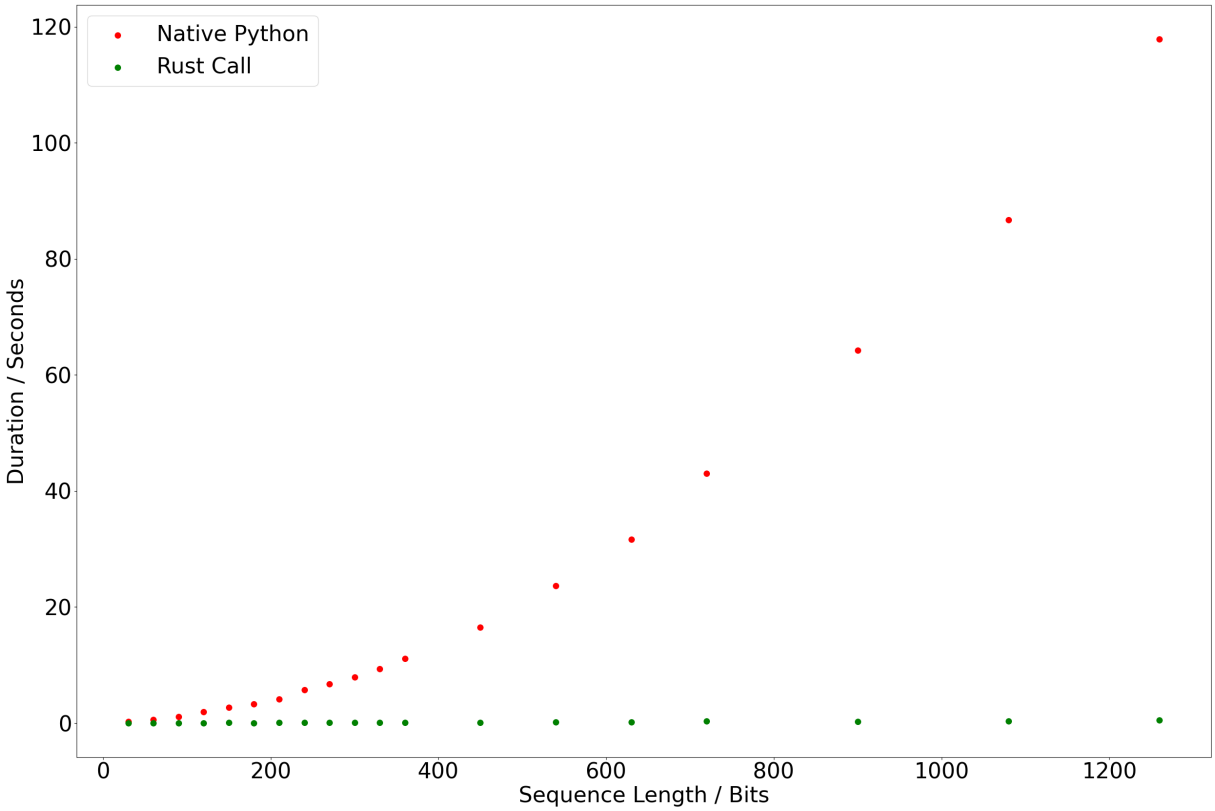


Figure B.2: Viterbi Decoding

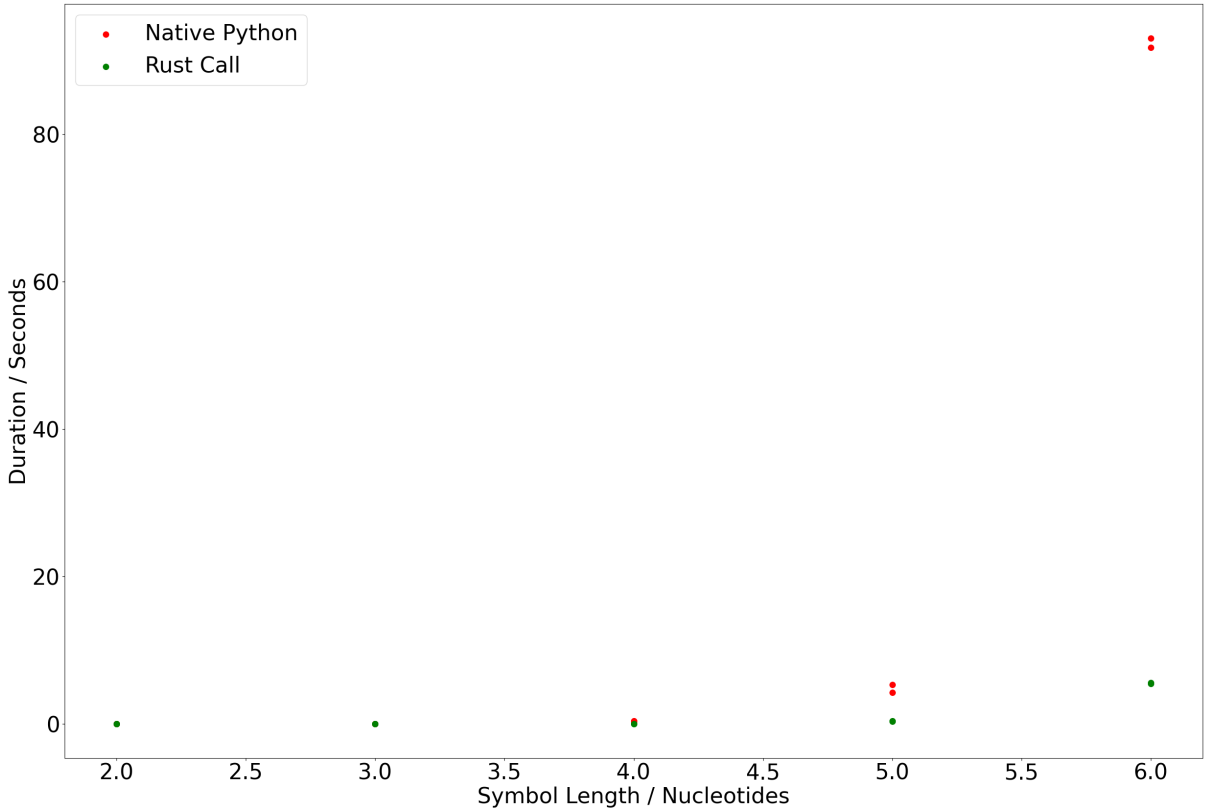


Figure B.3: Finite-state Machine Generation

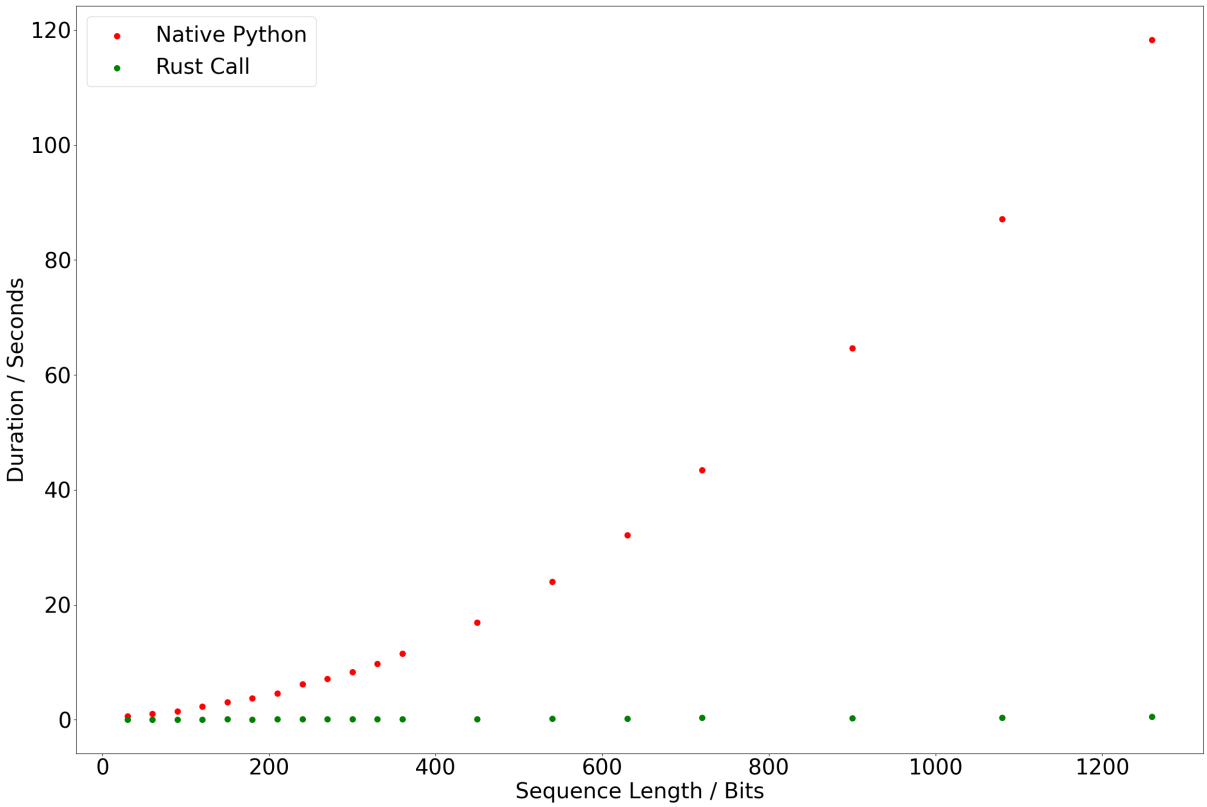


Figure B.4: Total Experiment Time

Appendix C

IDT gBlocks Results

An experiment was carried out using IDT's online gBlocks tool. A pathological DNA sequence is created and rejected by the tool, shown in figure C.1. The sequence is then encoded using the encoding scheme, and the resulting sequence is given to the tool again, at which point it is accepted, shown in figure C.2. The unencoded sequence was given a very high complexity score of 142.3. Encoding it reduced the complexity to 7.1.

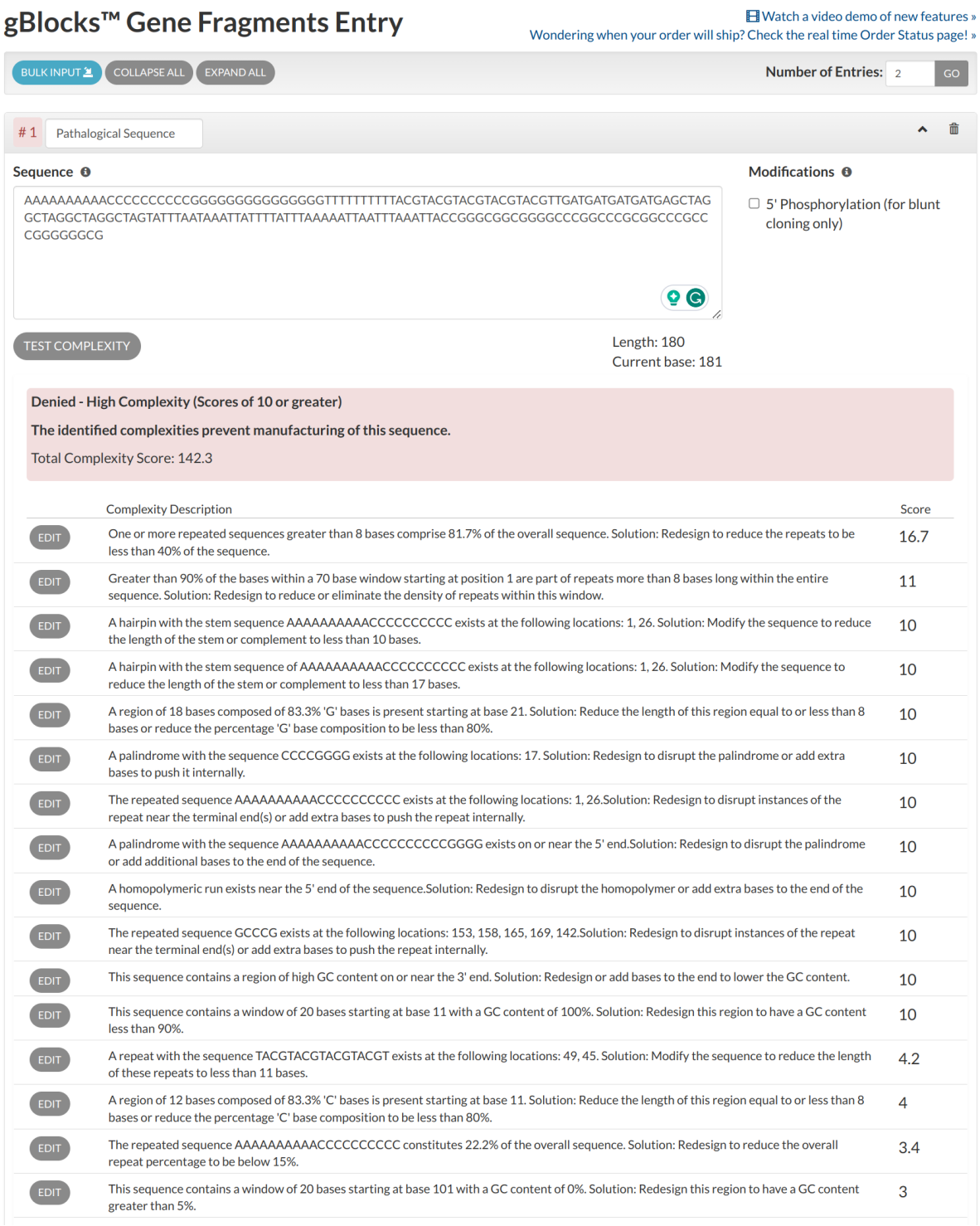


Figure C.1: The pathological sequence being given to IDT's gBlocks tool. It is rejected with a very high complexity score.

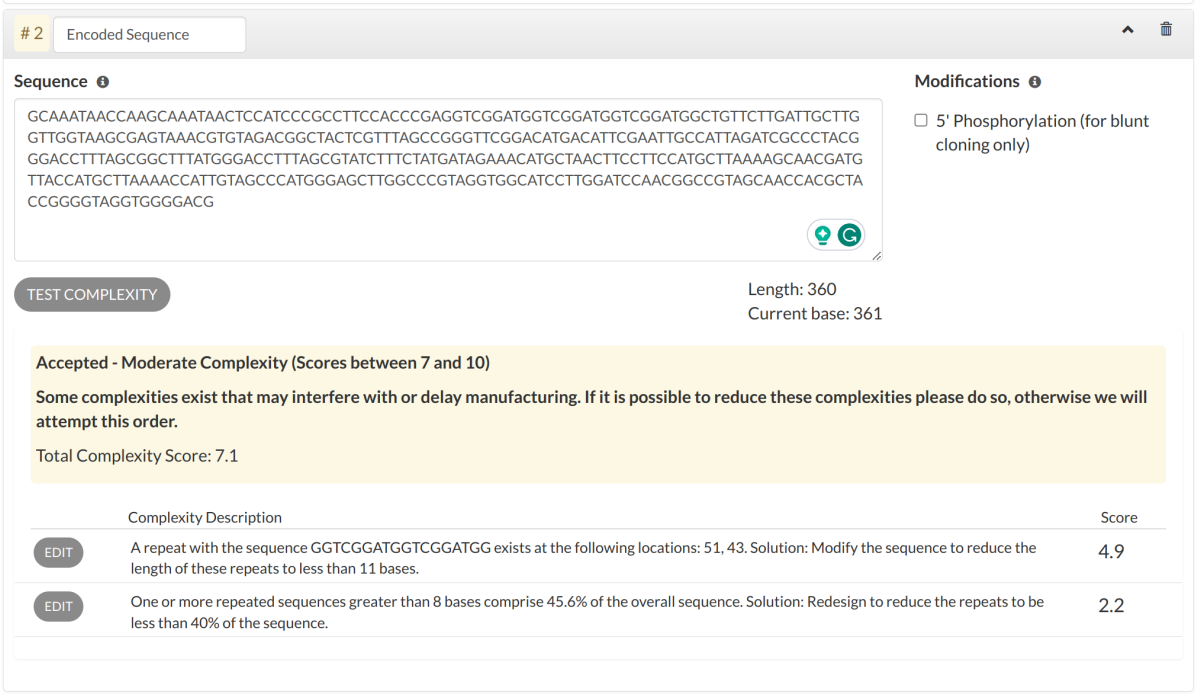


Figure C.2: The encoded version of the same pathological sequence. It is accepted with a much lower complexity score, but it is still not perfect - there are some manufacturer concerns which have not been addressed, but these are not sufficient to prevent synthesis.

Appendix D

Constraint-driven Convolutional Coding for Error Correction in DNA Storage

CONSTRAINT-DRIVEN CONVOLUTIONAL CODING FOR ERROR CORRECTION IN DNA STORAGE

Izer Onadim
Department of Computing
Imperial College London
izer.onadim19@imperial.ac.uk

Omer Sella
Department of Computing
Imperial College London
o.sella@imperial.ac.uk

Thomas Heinis
Department of Computing
Imperial College London
t.heinis@imperial.ac.uk

ABSTRACT

As rates of information production continue to increase exponentially, progressively denser data storage devices are needed. One promising approach is to store digital data on synthetic DNA. This option is appealing due to the incredibly high information density and durability of DNA. The process of storing digital data in nucleic acid requires a translation between binary digits and the quaternary alphabet of DNA. This mapping must take into account various biochemical constraints, whilst trying to minimise errors and maximise information density. Several mechanisms for such a mapping have been proposed and experimented with, including a finite-state machine (FSM) based, constraint-driven encoding by Sella *et al.* [1], who evaluated it on its ability to satisfy constraints whilst minimising coding overhead.

Here, we present how this encoding can be leveraged to correct certain categories of errors without any additional error-correcting codes (ECCs). We demonstrate that, when faced with substitutions in particular, the encoding scheme is highly effective at detecting and correcting errors. Performance on other categories of errors is also assessed. We additionally evaluate the different error-correcting capabilities of mappings produced by the scheme as they relate to hyperparameters such as symbol length, coding overhead and ability to meet biochemical constraints. Furthermore, this work proposes certain novel enhancements to the original encoding scheme and demonstrates that these changes allow the resultant encodings to account for previously unaddressed biochemical constraints. In support of these objectives, an efficient implementation of the amended version of Sella *et al.*'s encoding scheme is also contributed.

Keywords DNA Data Storage · Constraint-Driven Encoding · Convolutional Codes

1 Introduction

Humanity's rate of data production is growing exponentially [2]. This has triggered investigations into different options for denser storage media, including storing data on DNA. Owing to the relatively slow speeds of sequencing and synthesising DNA, it is likely to be utilised only for 'archival' storage purposes, in which data is written once and stored for many years or decades with a limited number of read accesses. Thus far, the primary applications for sequencing and synthesis technologies have come from the life sciences, in which very high accuracy is required; this has kept the cost of these procedures very high at several cents per nucleotide. In the field of data storage, the integration of error-correcting codes is commonplace, allowing for the accuracy requirements to be relaxed, and potentially decreasing the per-nucleotide cost to a sufficient degree to make DNA a viable archival storage medium.

In 2012, Church *et al.* experimented with storing digital information in DNA[1]; they wrote 650KB of data to DNA and read it back with relatively few errors [3]. The maximum data density of DNA is around 10^{21} bits per centimetre cubed[2], several orders of magnitude greater than the densest widely-used storage media [4, 5]. Synthesising and sequencing DNA involves many slow operations, which have only recently begun to be automated [6], so DNA latency is unlikely to catch up to that of electronic media[3]. This downside is offset somewhat by the concurrency of all DNA operations; sequencing (reading), amplification (copying) and synthesis (writing) are all highly parallel, making for an estimated throughput on the order of kilobytes per second [7].

In archival storage workloads, primary factors include density, durability, dollar cost per gigabyte read/written and energy cost at rest. DNA is highly durable; when stored in bone with no treatment to improve durability, it has an estimated half-life of approximately 520 years [8]. It has been seen to last considerably longer in cold, dark, and dry environments, as evidenced by the 2013 recovery of a horse genome from a bone trapped in permafrost for 700,000 years [9]. Current storage media, such as hard disk drives and

[1]Though they were not the first to do so, this was the first systematic experimentation with DNA as a potential storage medium

[2]The issue of finding a precise information density for DNA is difficult; one must make assumptions about the form in which the DNA is stored, whether it is double or single-stranded, and how tightly it is packed. The figure quoted is the maximum density that can be achieved when storing double-stranded DNA in a dried, crystal form, according to Zhirnov *et al.* [4]. However, as pointed out by Zhirnov *et al.*, the operations of reading, writing and copying DNA require other components, so it may be more realistic to assume an information density similar to that of a cell such as *E.coli* ($\sim 10^{19}$ bits/cm^3 [4]), which is still considerable.

[3]Based on the cell division of *E.coli*, a best-case read/write latency on the order of 100 µs per bit [4] can be estimated, but it is likely that electronic media will have achieved much lower latencies by the time automated sequencing/synthesis technologies approach this limit.

magnetic tape, are rarely given warranties of more than five or ten years. Moreover, DNA has a meagre cost when at rest, i.e. when not being accessed, since the conditions needed to store it safely are easily and cheaply achieved[4].

DNA is unlikely to suffer from obsolescence like other storage media. Floppy disks and VHS tapes, for example, are currently rarely used. The technology for reading these devices is no longer being improved and may one day be forgotten (or at least become very expensive and difficult to attain). DNA, the molecule encoding all genetic information, which has seen ever-increasing utilisation in medicine and biology, is unlikely to suffer the same fate. It seems reasonable to assume that for as long as there are biological humans with DNA, the ability to sequence and synthesise it is likely to be maintained and improved, especially with the advent of genetic engineering.

In order to store digital data as DNA, a function must be defined to map a string of binary data to a sequence of quaternary nucleotides. This mapping is known as an encoding, and it will often integrate indexing information, error-correcting codes, and possibly also primer sequences for random access into the resultant nucleotide sequence. This sequence is then synthesised into a strand of DNA which can then be stored, usually *in vitro*. After storage for an arbitrary length of time, the data encoded in the DNA can be accessed via DNA sequencing, for which there are a number of techniques. At this point, the quaternary string of nucleotides can be decoded back into binary data, and error-correcting codes can be applied to detect and correct any errors that have arisen. This sequence of steps is sometimes referred to as the DNA data storage "lifecycle".

When designing an encoding to map binary to quaternary data, there are many constraints that must be considered - it is usually insufficient to simply map each of the four two-bit sequences[5] to one of the four nucleotides. Such a naive encoding will result in a nucleotide sequence that is error-prone. This is because the various techniques for performing synthesis, amplification and sequencing experience higher error rates when writing/reading certain base sequences. An example of such a sequence is a homopolymer - a subsequence consisting of more than one consecutive repetition of a single base, e.g. AAAA or CCC. Homopolymers are prone to PCR[6] slippage, which can result in deletion errors, so homopolymers longer than a certain maximum length are avoided. There are many other similar constraints which an encoding may take into account.

An important consideration is that, as pointed out by Sella *et al.* [1], constraints are likely to change as techniques for reading, writing and copying DNA develop. Similarly, differing aims may lead to differing requirements; for example, when synthesising DNA for random access, a primer sequence is added to identify and extract different sections of stored DNA - this primer sequence would need to be reserved and not produced within the data. It is not difficult to imagine that different DNA storage applications in the future may need further reserved sequences. The result of this is that as constraints and applications change, the encodings used will become obsolete and will need to be updated. In response to this problem, an encoding generation mas proposed that can be used to create an encoding based on a given set of constraints. This will allow for different encodings to be generated using different constraint lists, catering for the requirements of various DNA data storage systems and simplifying the process of adapting to changes in the underlying technologies. A more detailed overview of the encoding generation mechanism is provided in section 2.2. This work builds upon the work of Sella *et al.* by first providing further analysis of the encoding generation scheme and then using this analysis to inform extensions and amendments to the encoding generator.

1.1 Thesis and Contributions

The principle aim of this work is to answer the question “Can an FSM-based, constraint-driven encoding scheme as presented by Sella *et al.* be leveraged to mitigate errors which arise in the archival DNA data storage life-cycle?".

The analysis performed herein clearly demonstrates that certain categories of errors can be successfully detected and corrected using such an encoding scheme, and examines the different error-correcting properties of various parameters that may be used by the scheme in the generation of a particular encoding. In support of this analysis, we also contribute an efficient implementation of the encoding scheme as it is described in [1]. Having assessed the error-correcting abilities of the standard encoding scheme, we go on to extend it in the hope of improving its ability to comply with biochemical constraints. We then analyse the enhanced scheme in a similar fashion to the original one and demonstrate that it can meet two previously unmet biochemical constraints without increasing coding overhead. Therefore, the contributions of this work can be summarised as follows:

1. An analysis of Sella *et al.*’s encoding scheme, which demonstrates its effectiveness as an ECC. This analysis explores how the parameterisation of the encoding generator affects the efficacy of the produced encodings, revealing which configurations lead to better error correction performance at different levels of coding overhead.
2. An extension of the encoding mechanism, including further analysis of the new version of the scheme. This extension presents two novel techniques for meeting biochemical constraints which the original scheme did not address.
3. A new, efficient implementation of the extended encoding generation scheme, as well as a framework for generating and experimenting with different encodings[7].

[4] As pointed out by Goldman *et al.* in ref. 10, the Global Crop Diversity Trust’s Svalbard Global Seed Vault would be a perfect environment to store DNA, despite having no permanent on-site staff, due to it being cold, dark and dry [11].

[5] The four permutations of two binary bits: 00, 01, 10, 11.

[6] Polymerase Chain Reaction. This is a fast and efficient process for amplification, i.e. making a very large number of copies of a DNA strand.

[7] The implementation is made freely available under an MIT Licence and can be accessed at `https://github.com/IzerOnadim/dna-storage`.

2 Preliminaries

2.1 Constraints

DNA Manufacturers stipulate a number of constraints on DNA sequences which can be synthesised to increase the chance of successful synthesis and lower the cost of the operation. Five of the most common constraints are addressed in this work.

Homopolymers are contiguous repetitions of the same base. This is a very common constraint, usually specified in terms of a "maximum run length", i.e. the maximum number of contiguous repetitions of the same base permitted. The encoding scheme is able to guarantee the absence of homopolymers longer than the specified maximum run length.

Short-Tandem Repeats can be thought of as a generalisation of homopolymers. They refer to any repeated subsequence. For example, 'ACGTACGT' is an STR as the subsequence 'ACGT' appears twice consecutively. There is no explicit handling of short-tandem repeats in the original description of the encoding scheme by Sella *et al.*, however, this work proposes a novel mechanism for partially mitigating violations of this constraint, which is described in section 5.2.

GC-Limits are upper and lower bounds placed on the GC content of a DNA sequence. The GC content is simply the proportion of a sequence which consists of Gs and Cs. For example, the sequences 'ACGT', 'CGCG' and 'ATAT' have GC contents of 50%, 100% and 0%, respectively. The encoding scheme can ensure that an encoded sequence has a GC content within a specified range of GC-limits.

GC-Variation is a limit on how much GC content can vary between two subsequences within a sequence. For example, a DNA manufacturer may stipulate that the GC content difference between the highest and lowest GC content subsequences of length 50nt[8] should be no more than 40%. That would mean that, in a particular DNA sequence, if the lowest GC content in a subsequence of length 50nt was 25%, the highest GC content in a different subsequence of the same length should be at most 65%. This constraint is not addressed by the Sella *et al.* encoding scheme, however, this work provides a novel mechanism for limiting the amount of GC-variation in an encoded sequence whilst maintaining error correction performance. This mechanism is described in section 5.1.

Reserved Subsequences are subsequences of nucleotides which should not appear in the output of the encoding. These subsequences may be reserved for a number of reasons, the common ones being that they are used as primers in PCR for sequencing or synthesis, or that they are used as indexing sequences in a random-access DNA system. The encoding scheme is able to avoid specified subsequences.

2.2 Constraint-driven encoding

The encoding scheme analysed and extended in this work was first presented by Sella *et al.* as a means of generating encodings based on a set of constraints. The rationale for creating such an encoding scheme is centred around the idea that as synthesis and sequencing technologies are innovated upon, the set of constraints that an encoding must take into account is likely to change; thus, an encoding targeting a prior set of constraints may become obsolete. In order to address this issue, the researchers presented what they describe as "a recipe for taking constraints and producing an appropriate encoding scheme". They explain that such a mechanism will allow the DNA encoding to be moved in lockstep with technological advances. Central to this "recipe" is the process of taking a set of constraints and producing a finite-state machine which describes how a DNA sequence should be produced from raw data. It is crucial to understand that this finite-state machine fully describes the encoding generated by the "recipe"; once the FSM has been defined, standard algorithms for convolutional coding and Viterbi decoding can be used to encode and decode any given sequence into DNA.

FSM construction occurs using the following process. First, the symbol length, L, is chosen. This can be thought of as the length of a "word" of encoded DNA. The aim is to find a sequence of DNA symbols of this length which does not violate any of the constraints in the constraint list. To this end, a simple assumption is employed; that none of the individual symbols of length L contains a violation of the constraints[9]. If the assumption holds, one can be assured that an individual symbol will not on its own contain a homopolymer or reserved subsequence violation, and so only the concatenations of these symbols need to be considered when checking for violations. To ensure that this assumption is true, it is necessary to set the symbol size to a value that is, firstly, less than or equal to the maximum length homopolymer permitted, and, secondly, strictly less than the length of the shortest reserved subsequence. Once an appropriate symbol size has been selected, the process of constructing the FSM can begin. The following algorithm is used; every possible concatenation of two symbols is considered, forming a connectivity matrix, with a row for each possible value of symbol one and a column for every possible value of symbol two. Each of these possibilities is checked against the entire list of constraints, and the matrix is populated with a one for every valid concatenation and a zero for every invalid (i.e. constraint-violating) one. Figure 1 shows a section of a possible connectivity matrix with a symbol length of three and only a single constraint to satisfy.

The algorithm proceeds to allocate a number of reserved bits (one at a time) at the front of the second symbol. It does this by "halving" the connectivity matrix; the matrix is essentially split into two by a vertical line, and the two halves are stacked on top of one another by taking an elementwise maximum. Each time this halving operation is performed, one bit of information (half a nucleotide) is relinquished as a reserved bit. This halving operation is repeated until all the cells in the connectivity matrix represent a valid concatenation, i.e. have a value of one. Once this is complete, the number of reserved bits will be known, and a set of possible values for the reserved bits of each symbol concatenation will have been produced. This set comes from all the instances in which two valid cells were halved onto each other; in this situation, the reserved bits from either cell can be

[8] Nucleotides.

[9] Strictly speaking, many of the symbols may violate the GC-limits constraint, however, unlike the constraints against homopolymers and restricted sequences, a violation of the GC-limits constraint may be fixed by concatenating two symbols together.

chosen as the new reserved bit values for the combined cell. Once this choice is made for each cell, FSM construction is complete. The FSM is structured in terms of states (symbol one, i.e. the previous symbol appended to the sequence) and inputs (the portion of symbol two which is not reserved). For each combination of state and input, there is a string of reserved bits which has been selected, and the next symbol to be appended to the sequence is determined by concatenating the input to the reserved bit string.

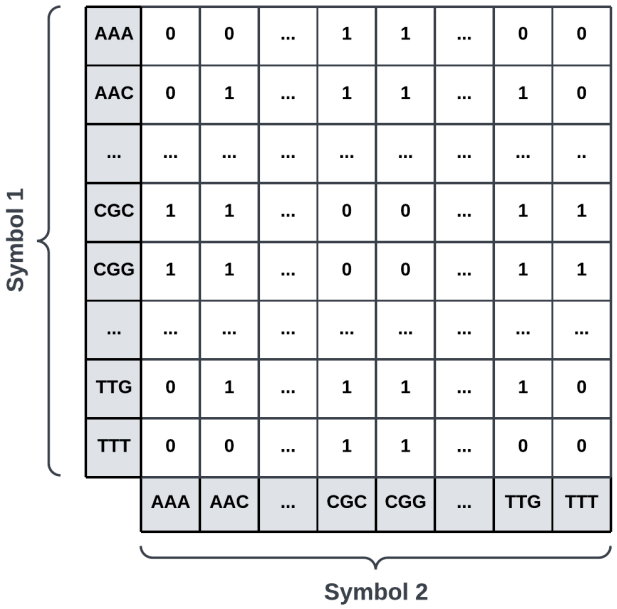


Figure 1: Transition matrix resulting from a enforcing a GC content between 30% and 70%

The halving mechanism represents an important departure of this work from the work done in [1]. In this work, we are interested in how the error-correcting capability of the produced encodings changes as the number of reserved bits is varied. Therefore, the number of reserved bits is defined as a separate parameter, which is controlled directly. If the number of reserved bits selected is insufficient to meet the given constraints, an error is simply returned. The advantage of this approach is that experiments can be run for different numbers of reserved bits without carrying out the process of halving the connectivity matrix multiple times to generate each FSM.

3 Methods

We begin by creating an efficient implementation of the encoding scheme along with an experimentation framework which can be used to test its error correction and constraint behaviour. The details of this implementation are not described in this article, but all software produced is made freely available. Experiments are then carried out using software simulation, the results of which are presented in section 4. Some enhancements are made to the encoding scheme and are then assessed through experiments.

3.1 Experiment Design

In creating an experiment design, our aim was to test the various hyperparameter configurations of the encoding scheme and compare their behaviour. To this end, it was defined that each "experiment" would measure the error and constraint behaviour of a particular configuration. To ensure the reliability of our results, each experiment consisted of multiple iterations, each of which was initialised with a different random seed. Within each iteration, a range of error rates was tested. For each of these error rates, a new FSM was constructed using the hyperparameter configuration to be assessed, next, a number of random strings were generated, encoded using the FSM, injected with errors at the specified error rate, and subsequently decoded. The input and output error rate of each of these encode/decode operations was recorded.

An experiment of this structure was carried out for each hyperparameter configuration of interest. The hyperparameters which were varied include:

- The Symbol length
- The number of reserved bits
- The mechanism for choosing the reserved bits
- The constraints set
- The type and distribution of injected errors

Symbol length and number of reserved bits are not independent of each other. The number of reserved bits must be strictly less than two times the symbol length (since the symbol length is in nucleotides, and each nucleotide represents two bits). Similarly, the number of reserved bits and the constraint set are not entirely independent. As the number of reserved bits changes with a fixed symbol size, the strictness of the constraints can also be varied. This is due to the fact that the greater the number of reserved bits, the stricter the constraint set can be. For example, with three out of eight bits (four nucleotides) reserved, GC content can be guaranteed to be within the range of 15% and 85%. If the number of reserved bits is increased to five out of eight, stricter GC-limits of 35% to 65% can be imposed.

3.2 Choice Mechanisms

As described in section 2.2, the encoding scheme works by splitting the encoded sequence up into symbols, and reserving a number of bits from the front of each symbol. These reserved bits can then be chosen from a pool of valid "candidates", which consists of all the possible allocations for the reserved bits that do not violate the constraints given in the constraint list.

A "choice mechanism" is then used to select one reserved bit string from the valid candidates. The process of selecting a reserved bit can be seen in figure 2. No choice mechanisms are prescribed by Sella *et al.* in [1], however, in his PhD dissertation Omer Sella mentions two possible mechanisms, which are random choice and GC-tracking. The GC-tracking choice mechanism works by selecting the reserved bit value that results in the GC content closest to 50%. This measure is not meant for ensuring GC content is within the given GC-limits; that is already guaranteed. Instead, this mechanism is intended to decrease GC-variation. Both of these choice mechanisms are assessed with regard to GC-variation and error correction.

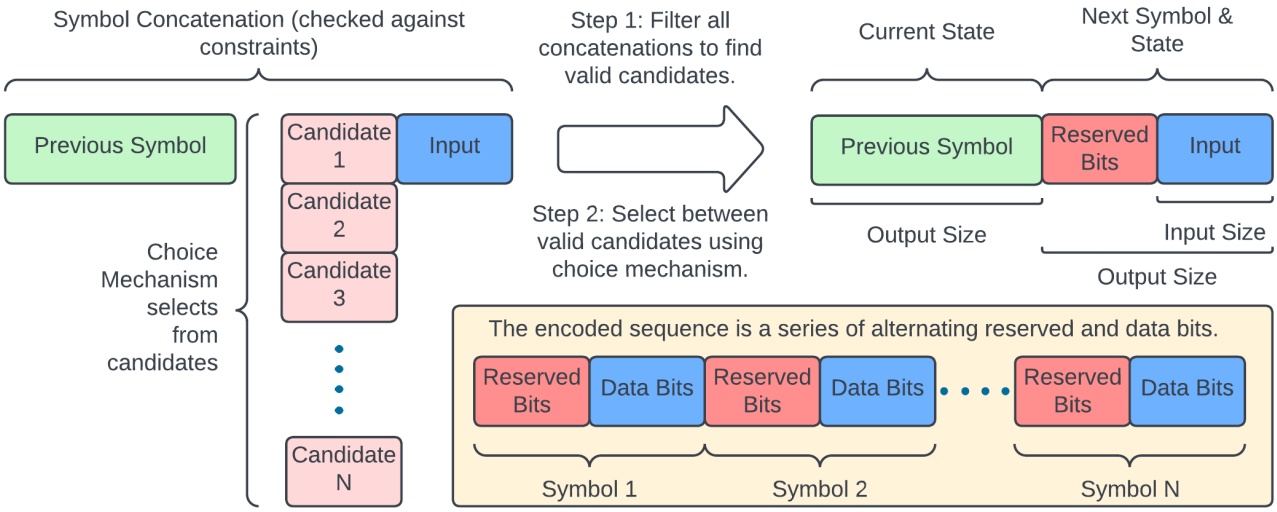


Figure 2: The process of selecting a valid candidate for the value of the reserved bits.

4 Results

4.1 Error Correction

The first set of configurations experimented with had a symbol size of four (four nucleotides, and therefore eight bits), and varied the number of reserved bits from three to five. Both GC-tracking and random choice were tested. The results of this experiment can be seen in figure 3.

From this figure, it can be seen that some configurations of the encoding scheme produce encodings with no capacity for error correction whatsoever, whilst others appear to be able to seriously reduce the prevalence of errors. The greatest factor affecting the output error rate appears to be the choice mechanism; random choice exhibits good error correction, whereas GC-tracking appears to have little to no ability to correct errors. When GC-tracking is used, the number of reserved bits seems to be mostly irrelevant, whereas, when random choice is used, there is a positive correlation between the number of reserved bits and error correction. This is unsurprising; the coding overhead is being increased, and in response, error correction is improving.

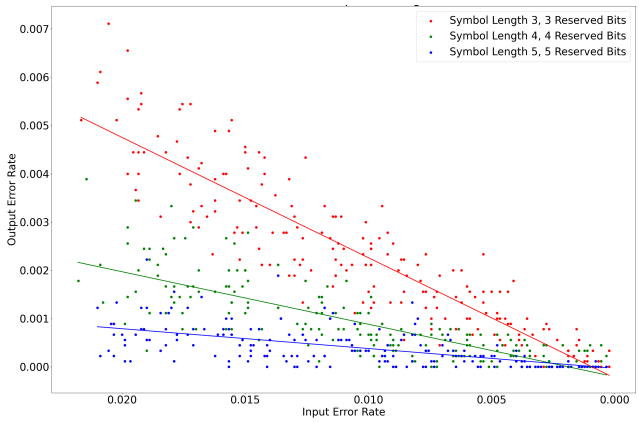


Figure 4: Comparison of different symbol lengths, with coding overhead kept constant. The random choice mechanism is used.

The next experiment compared the effect of symbol length on the output error rate. To try to isolate the changes caused by symbol size alone, each symbol length was tested with a constant coding overhead, i.e. the ratio of data bits to reserved bits was kept constant, at exactly half. Figure 4 demonstrates the results of this experiment.

This figure makes it clear that, providing the ratio of reserved bits is kept constant, larger symbol sizes result in better error correction. This comes with a serious trade-off, however, which is that both the FSM generation and Viterbi decoding algorithms have an exponential time complexity with respect to symbol size, putting a limit on how much this parameter can realistically be increased.

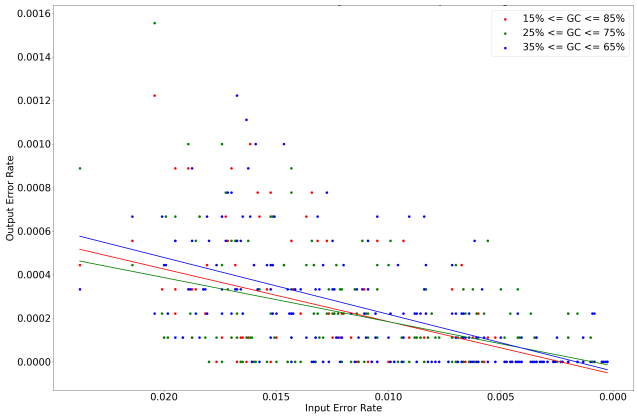


Figure 5: Varying constraint set strictness. The symbol length is four, the number of reserved bits is five, and the random choice mechanism is used.

Another interesting question is how much of a difference the strictness of the constraint set makes to the error correction performance. It may seem reasonable to expect error correction to degrade as the constraint set becomes stricter since there will be fewer valid candidates for the value of reserved bits, which may result in a smaller range of symbols being available for use, and as a result, more errors during Viterbi decoding. However, figure 5 shows that, for a reasonable set of constraints, this does not appear to be the case. It still

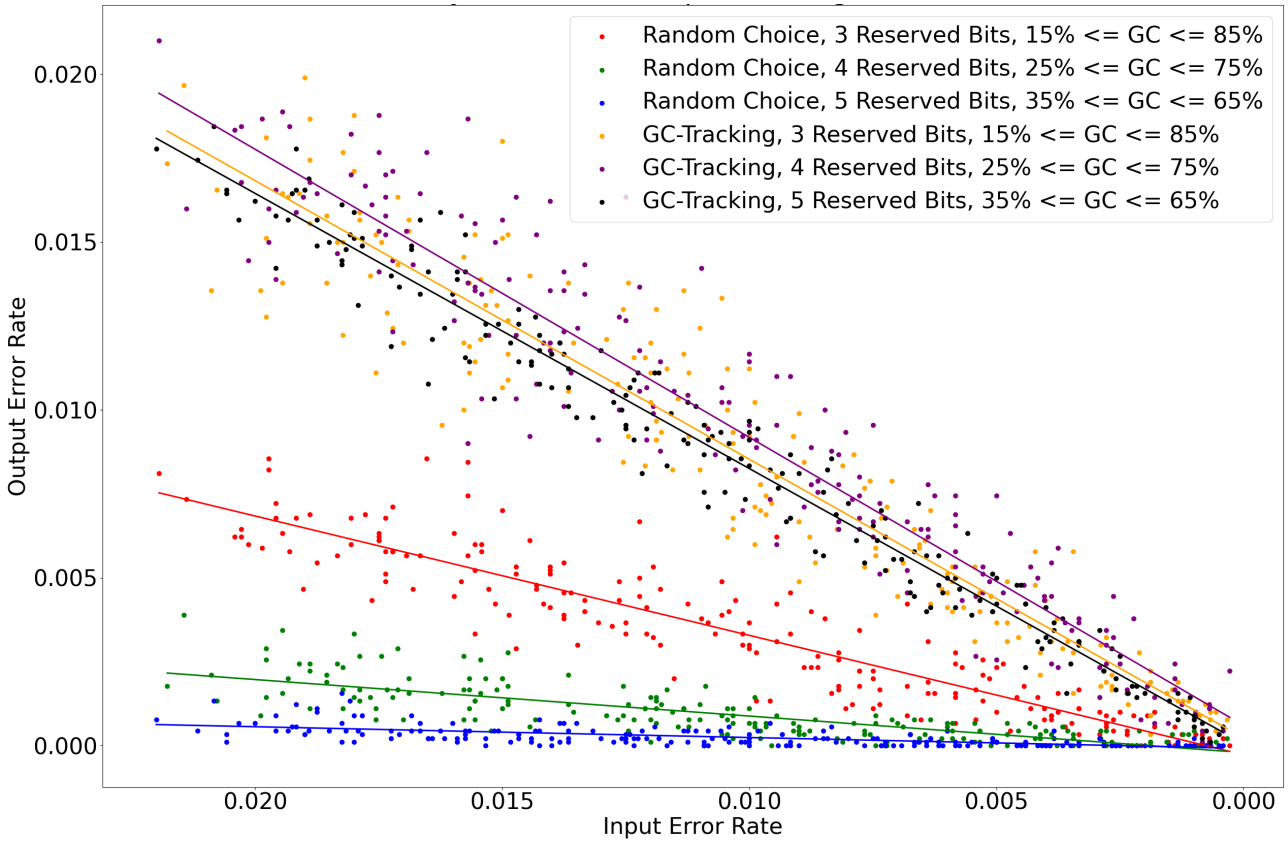


Figure 3: Input VS output errors with a symbol size of four. The choice mechanism and number of reserved bits are varied.

stands to reason that an extremely strict constraint set would degrade error correction, however, constraints such as those routinely stipulated by DNA manufacturers do not appear to be strict enough to affect error correction.

Lastly, the question of how the type and distribution of errors affect error correction is investigated. Since convolutional coding and Viterbi decoding are used by the encoding scheme, no resistance to insertions and deletions is expected; and this indeed turns out to be the case, with even a single insertion or deletion destroying the error correction of all encoding configurations. This is not as big a problem as it may seem since, due to the parallel nature of DNA operations, many identical strands can be synthesised at the same time, and therefore it is easy enough to simply discard DNA strands which are the wrong length, thereby cutting out the vast majority of insertions and deletions. If it is still necessary to correct insertions and deletions, a separate outer error-correcting code must be used.

Having established that insertions and deletions cannot be corrected by the encoding scheme, the question of error distribution is considered. Figure 6 demonstrates that, whilst short bursts of errors can be partially or entirely corrected, longer burst errors seriously degrade the error correction. As the length of the burst exceeds the symbol length, it becomes difficult or impossible to correct, which is expected.

These results, therefore, show that certain configurations of the encoding scheme produce encodings which are good at correcting substitution errors. The limitations of the encoding scheme have also been demonstrated in that it is unable to handle insertions, deletions, or long burst errors. When targeting substitution errors, it has been shown that symbol length should be increased as much as computing resources will allow, the number of reserved bits should be increased as much as coding overhead requirements permit, and the random choice mechanism should be used.

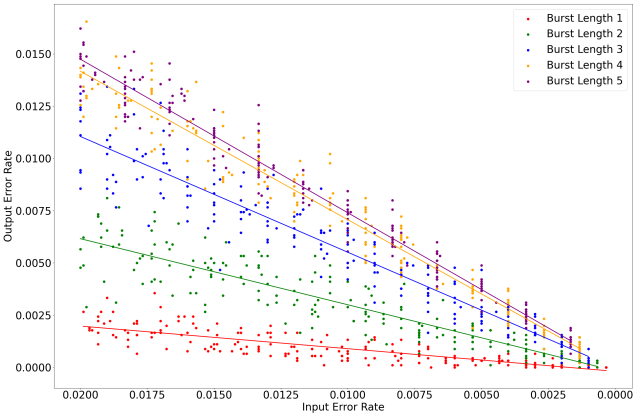


Figure 6: Varying constraint set strictness. The symbol length is four, the number of reserved bits is four, and the random choice mechanism is used.

4.2 Constraint Compliance

Having measured the error correction behaviour of the encoding scheme, the next step is to assess the constraint compliance of the generated encodings. As per the construction of the finite-state machine, compliance with GC-limits, max-

imum homopolymer length and reserved subsequences are guaranteed. However, no guarantees are provided regarding GC-variation or short-tandem repeats. The encoding scheme's performance on these factors is assessed so as to gauge whether there is any reduction in the violations of these constraints.

Firstly, GC-variation is measured with both the random choice mechanism and GC-tracking. The process for measuring GC-variation is simple; a window size is picked, which is the length of the subsequences that will be compared for their GC content, and each subsequence of this length has its GC content measured. The GC-variation is the greatest difference in GC content between any two of these subsequences. Since the overall GC content is guaranteed to be within the GC-limits, as the window size is increased, GC-variation will inevitably decrease, however, it is a desirable property to have a low GC-variation even with a relatively short window size, since this is a constraint many DNA manufacturers specify.

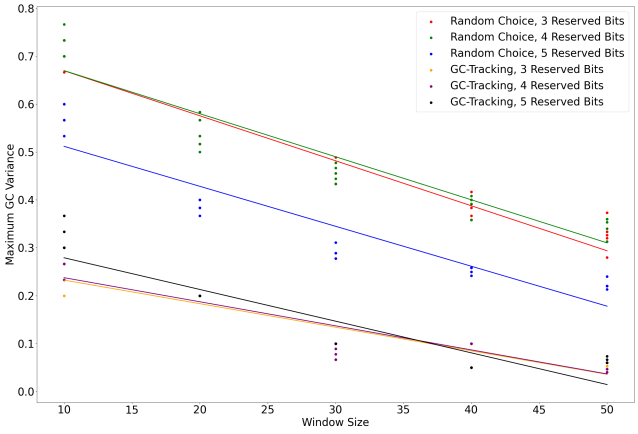


Figure 7: GC-variation as a function of window size (i.e. the length of the subsequence for which GC content is measured).

Figure 7 demonstrates the GC-variation results of an experiment involving many randomly generated strings. It is clear that GC-tracking is able to significantly decrease the GC-variation, as it was intended to do. An interesting observation to note is that when GC-tracking is used, the number of reserved bits does not seem to matter, at least within the range of three to five. Whereas, when random choice is used, allocating five reserved bits produces less GC-variation than three or four. This result firmly establishes that GC-tracking works as intended.

The encoding generation scheme has no method for mitigating short-tandem repeats. With that said, it is worth investigating whether the encodings have any effect on the prevalence of STRs. A method for measuring the occurrence of STRs is developed; first, a range of STR lengths is defined to be the range of interest - in this work, an STR length range of three to twenty is used. The string to be assessed is iterated through, and the number of consecutive repetitions of each of these lengths is calculated. For example, the string "ACGACGACG" would have an STR count of three at length three and zero for all other STR lengths. The string "ACGTACGT" would have an STR count of two at length four, and zero for all other lengths.

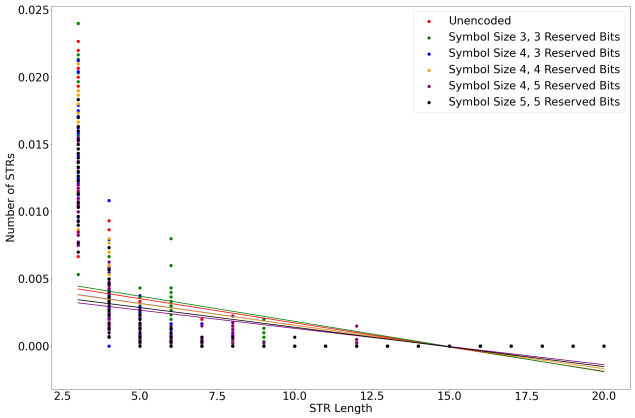


Figure 8: The prevalence of STRs of different lengths.

Figure 8 plots the prevalence of STRs at each length with different hyperparameter configurations. Before plotting, the number of STRs needs to be divided by the length of the whole sequence, since different hyperparameter configurations produce encoded strings of different lengths. This figure shows that no noticeable difference is made in the prevalence of STRs by the encoding, regardless of the configuration. The other interesting observation from this plot is that long STRs are very rare, and short ones are common. This suggests that simply reducing short STRs may yield a significant increase the constraint compliance of encoded sequences.

5 Extending the Encoding Scheme

Guided by the above results, this work proposes two novel extensions to the encoding scheme. The first is a new choice mechanism called randomised GC-tracking, which attempts to marry the error correction of the random choice mechanism with the GC-variation control of the GC-tracking choice mechanism. The second is a so-called "soft" constraint on STRs which is able to reduce the prevalence of STRs, but not eliminate them entirely.

5.1 Randomised GC-tracking

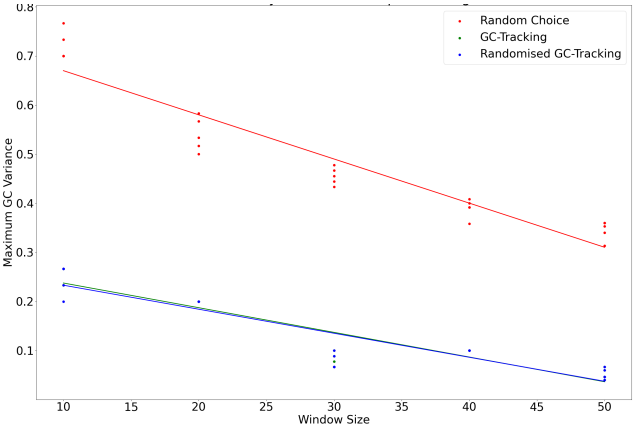


Figure 9: GC-variation as a function of window size. Randomised GC-tracking performs as well as standard GC-tracking. The symbol size is four, and the number of reserved bits is five.

The idea behind randomised GC-tracking is simple. It tests the GC content resulting from the use of each of the reserved bit candidates and records every candidate which produces the optimal GC content value. Optimal GC content, in this case, means as close as possible to 50%. It then picks randomly from this optimal set. This differs from standard GC-tracking, which simply selects the first candidate with the optimal GC content.

Figure 9 demonstrates that randomised GC-tracking exhibits the same GC-variation restricting behaviour as GC-tracking. Next, the error correction performance of randomised GC-tracking is assessed. Figure 10 demonstrates that randomised GC-tracking has far better error correction than GC-tracking, though it is still not quite as good as the random choice mechanism.

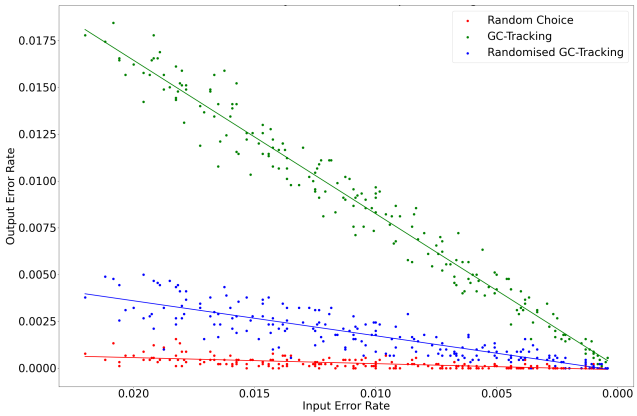


Figure 10: Error correction comparison of the choice mechanisms. The symbol size is four, and the number of reserved bits is five.

From this result, it can be seen that randomised GC-tracking is successful in its aims as it maintains all the GC-variation control of GC-tracking whilst recovering the majority of the error correction performance of the random choice mechanism.

5.2 Soft STR Constraints

This work partially mitigates STRs through the introduction of a "soft" constraint on them. This is referred to as a "soft" constraint since it does not preclude the occurrence of STRs but merely decreases their prevalence. The soft constraint works through the specification of a lower and upper bound, which represent the inclusive range of STR lengths which the encoding scheme will attempt to filter out. A limitation of this is that the upper bound must be less than or equal to the symbol length. The soft constraint then invalidates all the reserved bit candidates which lead to an STR of a length within the specified range. This is not guaranteed to eliminate all STRs of a particular length since only the concatenation of two symbols is inspected at a time, so if the inspection of three consecutive symbols needs to be inspected to detect the STR, it will be missed. Table 1 demonstrates how an STR can be missed by this mechanism by taking as an example an STR of length three and a symbol size of length four.

Figure 11 demonstrates that seriously decreases the prevalence of STRs, but does not eliminate them entirely. In this figure, it can be seen that STRs of length three have been completely removed, whereas STRs of lengths four and five have been reduced. STRs longer than five (which is the symbol length in this example) are unaffected.

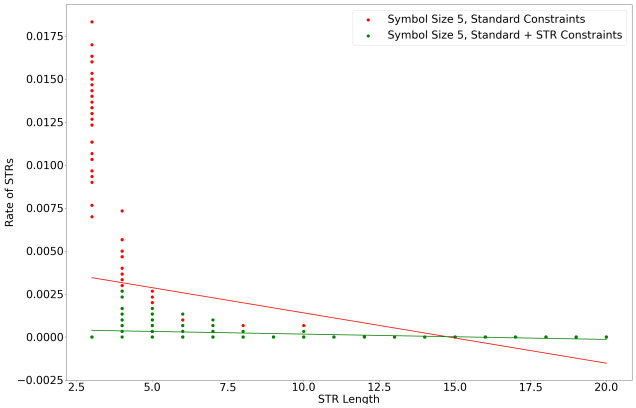


Figure 11: The prevalence of STRs with and without the soft STR constraint. The symbol size is five.

Since the enforcement of the soft STR constraint is tied to symbol length, an interesting comparison is how symbol length affects the efficacy of the constraint in reducing STRs. Figure 12 shows the relationship between STR occurrence and symbol size. This figure clearly shows that larger symbol sizes result in better STR prevention. There are two reasons for this. The first is that STRs longer than the symbol length cannot be detected by this mechanism, so increasing the symbol length improves STR prevention. The second is that, as the symbol size increases, it becomes more difficult (or impossible) for an STR of a particular length to span across three symbols, and therefore, detection is improved (or guaranteed). For example, with a symbol length of five, all STRs of length three are guaranteed to be detected and prevented since there is no way for an STR of length three (which must contain at least six nucleotides, since it is a consecutive repetition of a subsequence of length three) to avoid detection in a concatenation of two symbols of length five (since there is no way for six nucleotides to span across three symbols of length five). This can be intuitively understood by observing that the situation in table 1 in which an STR of length three was missed would not be possible with a symbol length of five.

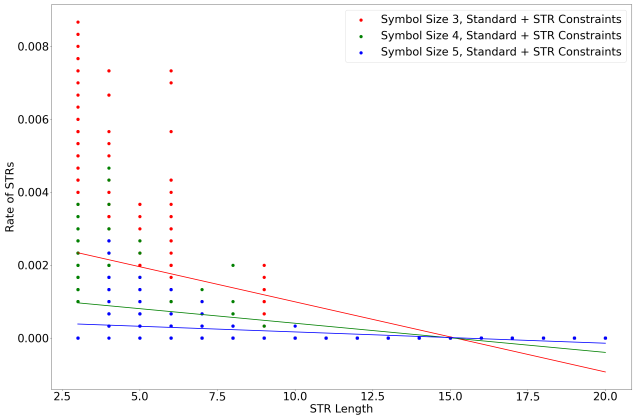


Figure 12: A comparison of STR prevalence at different symbol sizes.

Another interesting question is whether this novel soft constraint on STRs affects error correction. Figure 13 demon-

Sequence	S1	S2	S3	Detected & Prevented?
ACGACGTATGCG	ACGA	CGTA	TGCG	Prevented by disallowing S1 + S2
GACGACGTATGC	GACG	ACGT	ATGC	Prevented by disallowing S1 + S2
CGACGACGTATG	CGAC	GACG	TATG	Prevented by disallowing S1 + S2
GCGACGACGTAT	GCGA	CGAC	GTAT	Not detected as no two symbols contain an STR
TGCGACGACGTA	TGCG	ACGA	CGTA	Prevented by disallowing S2 + S3
ATGCGACGACGT	ATGC	GACG	ACGT	Prevented by disallowing S2 + S3
TATGCGACGACG	TATG	CGAC	GACG	Prevented by disallowing S2 + S3

Table 1: Each of the sequences shown is a rotation of the first sequence, which contains the STR “ACGACG". Assuming a symbol length of four, the sequence will be spread out across three symbols. Since the STR checking happens at the concatenation site of two symbols, if the inspection of three consecutive symbols is required to detect the STR, it will not be detected. Therefore some STRs will get through.

strates that this is not the case by plotting the error rate of three different configurations with and without the soft STR constraint. In each case, it is clear that the error profile is almost identical in the cases with and without the constraint, which indicates that this additional constraint improves constraint compliance for "free", at least with regard to error correction.

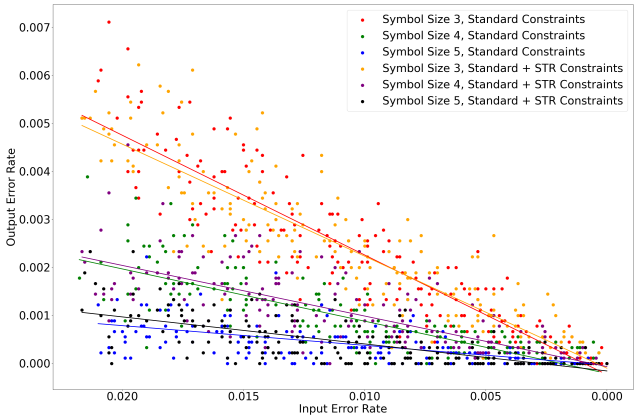


Figure 13: Error rates at different symbol sizes with and without the soft STR constraint.

6 Discussion and Evaluation

The encoding generation scheme presented in this work simultaneously enforces given biochemical constraints and also provides protection against substitution errors. However, it is unclear how much of the overhead is being used for constraint compliance and how much is being used for error correction. To assess this, a comparison is provided between a standard $1/2$ convolutional code and an encoding generated by the encoding scheme with the same overhead.

Figure 14 shows the results of this comparison. Interestingly, a symbol size of one with one reserved bit seems to outperform the $1/2$ convolutional code by a small margin. This is surprising due to how similar these encodings are in structure - the only difference is likely to be the random allocation of the reserved bit, which may have produced a trellis which is more effective at correcting nucleotide errors. As expected, longer symbol sizes greatly improve error correction, but at the cost of slower encoding/decoding. These results suggest that the encoding scheme provides superior error correction to the $1/2$ convolutional code despite having the same overhead. Another thing to note is that there seems to be very little difference between the error correction performance with no constraints and with the default constraint set, which indicates that avoiding violations of the default set of biochemical constraints does not degrade error correction.

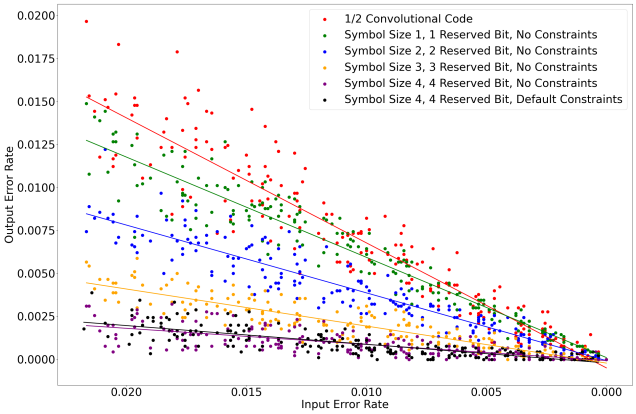


Figure 14: Comparison of $1/2$ convolutional code with encodings generated by encoding scheme. In all cases, exactly half the encoded sequence is used for error correction, so coding overhead is constant.

Encoding performance is also an important factor for evaluation. The two most important factors affecting the performance are symbol size and sequence length (i.e. the length of the sequence to be encoded into DNA).

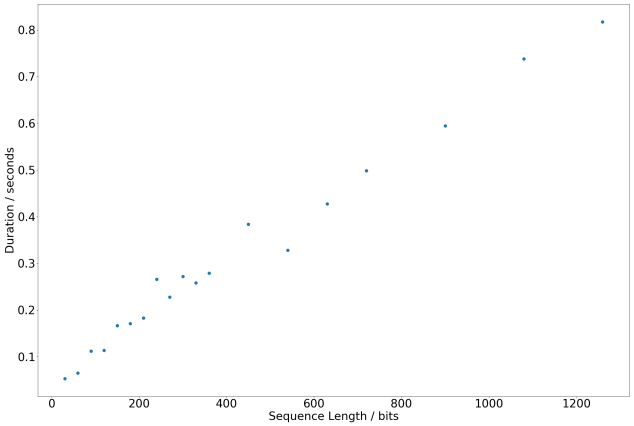


Figure 15: A profile of a round trip with respect to the sequence length.

The encoding scheme consists of three primary sections: FSM generation, convolutional coding, and Viterbi decod-

ing. To evaluate the performance of the encoding scheme, a test of the "round trip" time is carried out, which involves one FSM generation operation, followed by a convolutional coding and Viterbi decoding operation.

Figure 15 shows the relationship between run duration and sequence length, which is linear. This result is to be expected since FSM generation has constant time complexity with respect to sequence length, whereas convolutional coding and Viterbi decoding are both linear in terms of sequence length. This result demonstrates that much longer sequence lengths could be encoded/decoded before this becomes a limiting factor; sequence lengths of over 1000 nucleotides took less than a second to encode and decode.

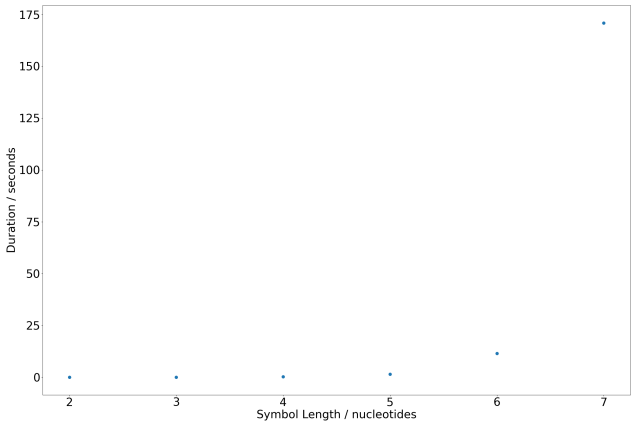


Figure 16: A profile of a round trip with respect to symbol size.

Figure 16 demonstrates the relationship between system performance and symbol length, which is exponential. Again, this is expected since FSM generation and Viterbi decoding both have exponential time complexity with respect to symbol length. This demonstrates that symbol length will be a limiting factor for performance if it is pushed beyond six.

7 Conclusion and Future Work

This work has demonstrated the efficacy of the encoding generation scheme described by Sella *et al.* in [1] as an ECC. It has clearly shown that encodings generated by the scheme are effective for correcting substitution errors in DNA storage, which is a novel result. The limitations of the encoding scheme's error correction with regard to insertions, deletions and burst errors have also been shown. Furthermore, different hyperparameter configurations which can be used to generate encodings have been assessed, and recommendations have been made regarding which configurations are suitable depending on the priorities and limitations of a user. It has also been demonstrated that error correction can be achieved by encoding without compromising compliance with biochemical constraints. Novel enhancements have been proposed to improve the constraint compliance behaviour of the encoding scheme, and then evaluated with respect to both their target constraints and their effect on error correction. Finally, an efficient implementation of the extended encoding scheme has also been contributed.

Many state-of-the-art approaches to DNA storage include one encoding step for addressing constraints and another for error correction. The encoding scheme described in this work is able to address both of these aims, however, it is not equally effective against all types of errors. Therefore, further work could be carried out to integrate dedicated error-correcting codes into the encoding scheme to improve its error correction. An approach which seems likely to yield good results is to keep the convolutional encoding scheme as the only inner code and add an outer code for better insertion/deletion performance. Prior work on convolutional codes has established that certain alterations to the Viterbi algorithm be made to allow for the correction of deletion errors [12]. It may be possible to integrate this enhanced version of the Viterbi algorithm into the encoding generation scheme, which may perhaps allow it to correct some deletion errors without the need for an outer code.

Future work also includes finding better ways of selecting the values of the reserved bits. This work demonstrated that the method for selecting reserved bits makes an enormous impact on the error-correction ability of the encoding, so it seems likely that there are further ways to decrease the error rate by varying this parameter. It may be possible to select the reserved bits in a more considered way, such that some error-correcting information is stored in the reserved bits, and subsequently made use of during decoding.

References

[1] Omer S. Sella, Amir Apelbaum, Thomas Heinis, Jasmine Quah, and Andrew W. Moore. Dna archival storage, a bottom up approach. In *Proceedings of the 13th ACM Workshop on Hot Topics in Storage and File Systems*, HotStorage '21, page 58–63, New York, NY, USA, 2021. Association for Computing Machinery.

[2] David Reinsel-John Gantz-John Rydning, J Reinsel, and J Gantz. Data age 2025: The digitization of the world from edge to core. *International Data Corporation & Seagate*, 16, 2018.

[3] George M. Church, Yuan Gao, and Sriram Kosuri. Next-generation digital information storage in dna. *Science*, 337(6102):1628–1628, 2012.

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

[5] Martin G. T. A. Rutten, Frits W. Vaandrager, Johannes A. A. W. Elemans, and Roeland J. M. Nolte. Encoding information into polymers. *Nature Reviews Chemistry*, 2(11):365–381, Nov 2018.

[6] Christopher N. Takahashi, Bichlien H. Nguyen, Karin Strauss, and Luis Ceze. Demonstration of end-to-end automation of dna data storage. *Scientific Reports*, 9(1):4998, Mar 2019.

[7] Luis Ceze, Jeff Nivala, and Karin Strauss. Molecular digital data storage using dna. *Nature Reviews Genetics*, 20(8):456–466, Aug 2019.

[8] Morten E. Allentoft, Matthew Collins, David Harker, James Haile, Charlotte L. Oskam, Marie L. Hale, Paula F. Campos, Jose A. Samaniego, Thomas P.M. Gilbert, Eske Willerslev, Guojie Zhang, R. Paul Scofield, Richard N. Holdaway, and Michael Bunce. The half-life of dna in bone: measuring decay kinetics in 158 dated fossils. *Proceedings of the Royal Society B: Biological Sciences*, 2012.

[9] Ludovic Orlando, Aurélien Ginolhac, Guojie Zhang, Duane Froese, Anders Albrechtsen, Mathias Stiller, Mikkel Schubert, Enrico Cappellini, Bent Petersen, Ida Moltke, Philip L. F. Johnson, Matteo Fumagalli, Julia T. Vilstrup, Maanasa Raghavan, Thorfinn Korneliussen, Anna-Sapfo Malaspinas, Josef Vogt, Damian Szklarczyk, Christian D. Kelstrup, Jakob Vinther, Andrei Dolocan, Jesper Stenderup, Amhed M. V. Velazquez, James Cahill, Morten Rasmussen, Xiaoli Wang, Jiumeng Min, Grant D. Zazula, Andaine Seguin-Orlando, Cecilie Mortensen, Kim Magnussen, John F. Thompson, Jacobo Weinstock, Kristian Gregersen, Knut H. Røed, Véra Eisenmann, Carl J. Rubin, Donald C. Miller, Douglas F. Antczak, Mads F. Bertelsen, Søren Brunak, Khaled A. S. Al-Rasheid, Oliver Ryder, Leif Andersson, John Mundy, Anders Krogh, M. Thomas P. Gilbert, Kurt Kjær, Thomas Sicheritz-Ponten, Lars Juhl Jensen, Jesper V. Olsen, Michael Hofreiter, Rasmus Nielsen, Beth Shapiro, Jun Wang, and Eske Willerslev. Recalibrating equus evolution using the genome sequence of an early middle pleistocene horse. *Nature*, 499(7456):74–78, Jul 2013.

[10] 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, Feb 2013.

[11] The Global Crop Diversity Trust. Svalbard global seed vault, 2023. (accessed 21 Jan 2023).

[12] Hugues Mercier and Vijay K. Bhargava. Convolutional codes for channels with deletion errors. In *2009 11th Canadian Workshop on Information Theory*, pages 136–139, 2009.