Delft University of Technology
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft Institute of Applied Mathematics

Constrained Error-Correcting Codes for DNA-Based Storage Systems

(Dutch title: Beperkte foutcorrigerende codes voor DNA-gebaseerde opslagsystemen)

Thesis submitted to the
Delft Institute of Applied Mathematics
in partial fulfillment of the requirements

for the degree

BACHELOR OF SCIENCE
in
APPLIED MATHEMATICS

by

F.J. Laseur

Delft, Nederland
December 2021

BSc thesis APPLIED MATHEMATICS

"Constrained Error-Correcting Codes for DNA-Based Storage Systems"
(Dutch title: "Beperkte foutcorrigerende codes voor DNA-gebaseerde opslagsystemen)"

F.J. Laseur

Delft University of Technology

December, 2021 Delft

Preface

During my years at the TU Delft I have learned a lot about Mathematics. I feel a strong personal motivation on its applications in our future society. It drives me to involve the outside world in my field of expertise, and not just limiting myself to discussing mathematical topics with peers. I am driven to encourage people to understand and contribute to technical matters, to reduce the gap between science and society. My final thesis in high school contained a comprehensible introduction to the theory of relativity. Last year I designed, built and controlled a scalable Hyperloop system together with 35 students. We organized and participated in a world wide competition that contributes to the sustainable infrastructure of the future. Science should not be something exclusive or unmanageable. Once more people start understanding, more people will be interested in science, instead of discouraged by it. I consider our profession widely applicable. Mathematics determines our daily life in many respects. The analytically based mindset we are taught is of great value in optimizing complex structures.
The courses Optimization and Decision Theory opened my eyes on the direct benefit we can enjoy of mathematics. I wanted to solve a problem for my Final Thesis, that could bring our daily life to a next level. The field of Coding Theory has great potential to achieve this goal. I soon got in touch with Dr. Ir. Jos Weber. I find his broad knowledge of various fields impressive. His career at the TU Delft, at several departments, starts and ends with Mathematics. I am very glad to contribute in his ongoing research on efficient ways to store data as DNA, and will follow it closely in the future. Also Dr. Joost de Groot was very much involved in the process. I enjoyed the aggregated resources of two of TU Delft's most professional and experienced teachers. I am very thankful and it was my pleasure to work on this innovative research.

Florine Laseur,
November 2021

Abstract

Humanity is producing data at exponential rates. Solutions need to be found in order to accommodate the bytes of today and the future in an efficient and environmentally friendly way. DeoxyriboNucleic Acid, well-known as DNA, is a suitable destination for this major collection of data, as it is a high-density storage medium with long storage endurance [1]. In the field of research about DNA-Based Data Storage Systems, people examine quaternary codes. The considered *DNA-codes* consist of equi-length *DNA-words* over the Alphabet $\{0, 1, 2, 3\}$ representing the available DNA building blocks or *nucleotides* $\{A, T, C, G\}$ standing for Adenine, Thymine, Cytosine and Guanine respectively. Combinatorial constrained coding is introduced to avoid DNA patterns prone to *sequencing errors*, that might occur when archival data is read from DNA-blocks [2]. If an error occurs anyway, an imposed minimum Hamming distance d that a DNA-code satisfies, enforces error correction and detection capabilities [3]. Methods are considered to optimize the number of DNA-words that can be stored in a DNA-code satisfying these restrictions.

This research explores DNA-codes of which the DNA-words adhere to a fixed GC content referred to as the GC-weight w, as well as the no-runlength constraint $r = 1$ that will not allow adjacent symbol repetition. By increasing distance d, we build on the findings of Van Leeuwen [4] and Vermeer [5], who created DNA-codes satisfying $d = 2$ for $r = 1$ and $r \geq 1$ respectively. The maximum size of a DNA-code satisfying $d = 3$ among other constraints is proven to be 12. Furthermore, we cover the case $d = n$ where the length n of the DNA-words and the minimum distance d the DNA-code satisfies are equal. We derive maximum sizes of these DNA-codes for all possible GC-weights w. Finally, we present algorithms that create DNA-d codes satisfying $d = 3$ and $d = 4$ and succeed in improving sizes of the DNA-codes created by the algorithms designed by Limbachiya et. al. [6] and Van Leeuwen [4].

Contents

1 Introduction

1.1 Motivation

DNA Based Data Storage is a promising technique that, if conducted correctly, could solve contemporary issues in Data storage. Our digital universe contains around 44 zettabytes of data [7]. That is roughly 400 times more bytes than grains of sand on earth [8]. DNA is an attractive medium to densely archive this digital information that continues to accumulate [9]. Writing and reading DNA is costly, but has great potential for archival data storage, due to it's high density [2] and longevity [6] compared to conventional digital data storage. Erlich et. al.[10] explored an architecture which approaches a theoretical maximum storage capacity of 215 petabyte per gram of DNA. Storing a single petabyte would take over 1.5 million CD-ROM discs. Furthermore, we can recover DNA of extinct species: 300,000 year old mitochondrial DNA from bears and humans has been recently decrypted [11]. DNA is suitable for non-volatile storage of information.
An expert in the field of coding theory, Kees A. Schouhamer Immink, is involved in the journey of data storage from the beginning: From the invention of the CD in 1982, to the introduction of DNA Based Data Storage Systems. He states: "You can not only read but also synthesize DNA nowadays. No long strands can be created, like in our genes, but pieces of at most 100 elements. If you utilize this technique efficiently, you can store the entire content of the internet in a glass of water."[12]
Research on constraints is being done in order to come up with efficient quaternary code designs to optimize this promising technique.

1.2 Thesis statement

Clearly, DNA is a promising host for our major collection of data due to its extremely dense storage capacity and outstanding integrity over long term storage. Techniques examine how to physically store DNA and maintain its stability over time. This requires coding schemes that can avoid error-prone patterns when writing and reading DNA-codes. We examine error sources and obtain that DNA-words should maintain an imposed ratio of G and C nucleotides [13], referred to as the GC-weight w. The more equal the distribution between A, T and G, C nucleotides within all DNA-words, the more balanced a DNA-code. This comes down to a GC-weight as close to half the DNA-word length n as possible: $w = \frac{n}{2}$.
Restricting the maximum homopolymer run r or adjacent repetition of symbols in the DNA-code also affects the error probability [6]. DNA-codes that satisfy the no-runlength constraint $r = 1$, hence contain no adjacent repeated nucleotides, are known to be less error prone. Another constraint to DNA-codes is the Hamming distance d: the least number of positions in which all DNA-words in the code differ from each other. The power of the Hamming distance d a DNA-code satisfies is that by improving it we can not only detect, but also correct more errors in falsely received DNA-words.
This thesis contributes to the research to efficient quaternary DNA-code design by creating and analyzing DNA-codes meeting given restrictions. Building on existing research [4] [5], we answer the research question:

"Can we create DNA-codes and therefore determine upper bounds and lower bounds for the size of DNA-codes satisfying $d > 2$?".

We determine the maximum size of a DNA-code satisfying $n = 4$, $w = 2$, $d = 3$. We also completely cover the case where we impose $d = n$ and define the maximum size of DNA-codes satisfying this constraint for all possible weights w. We obtain (lower bounds for) the maximum size of DNA-codes satisfying $d = 3$ and $d = 4$ by presenting algorithms generating valid DNA-codes. The higher purpose is to use DNA Data Storage in the most efficient way. This translates to housing the largest possible number of DNA-words satisfying specified constraints.

1.3 Organisation of the Thesis

We present an overview of the remaining content of this thesis by Chapter.

Chapter 2: Prerequisites: This chapter discusses an explanation of DNA and the basic concepts of DNA Based Storage Systems. Furthermore, we present basic concepts of coding and provide definitions for a DNA-code and a DNA-d code.

Chapter 3: The impact of constraints: In this Chapter, we elaborate on constraints and the impact of changing them on restricted code generation. Also, we discuss two decoding mechanisms: error detection and error correction.

Chapter 4: The maximum size of a DNA-3 code: This chapter provides a proof that the maximum size of a DNA-code satisfying $n = 4$, $w = 2$ and $d = 3$ is 12. This result is invariant under changing the runlength constraint, hence holds for each imposed r. We conclude $B_r(4, 2, 3) = 12$.

Chapter 5: The maximum sizes of DNA-n codes: This chapter provides the determination of the maximum size of DNA-codes satisfying $d = n$. This result is invariant under changing the runlength constraint, hence holds for each imposed r. We determine $B_r(n, w, n)$ for every possible GC-weight w: We find lower bounds for this value by creating valid DNA-codes satisfying the imposed constraints. Subsequently we derive contradictions when trying to improve these lower bounds.

Chapter 6: Algorithms generating DNA-d codes improving existing DNA-d code sizes: This chapter presents and discusses algorithms from existing research [4] [6]. Moreover, original algorithms are presented that generate valid DNA-codes that adhere to a universal DNA-word length n, the no-runlength constraint $r = 1$, a balanced GC-weight $w = \lfloor \frac{n}{2} \rfloor$ and an imposed distance $d = 3$ and $d = 4$ respectively. The algorithms succeed in improving DNA-d code sizes of existing research.

Chapter 7: Conclusions and Recommendations : This chapter provides an overview of the results of this thesis. Furthermore, we pose suggestions for future contribution to the optimization of DNA Based Data Storage. Recommended analytical research as well as suggested improvements of the presented algorithms are described.

2 Prerequisites

DNA Based Storage Systems are suited for future accommodation of the ever accumulating collection of data [9]. In order to archive information as DNA, we have to carry out quaternary code generation. This chapter describes the basic concepts as well as the stages of DNA Based Data Storage.

2.1 What is DNA and how do we read and write it?

DeoxyriboNucleic Acid, known as DNA, hosts genetic information of cells. The structure of a DNA molecule is a double-helix polymer[14]: two DNA-strands wound to a spiral. Each strand is a chain of nucleotides. Each nucleotide partially consists of one of the four bases Adenine, Thymine, Cytosine and Guanine, after which the nucleotides are named. We will use abbreviations A, T, C and G respectively. The strands are held together in the spiral shape by hydrogen bonds between the bases: Adenine bonds with Thymine only and likewise Cytosine only bonds with Guanine. Due to this strong connection between strands, the configuration of a DNA molecule is very stable. This is beneficial for long term storage in our application of DNA as a medium for Data Storage.
When a DNA Based Storage System is used, data cycles through the following stages [6]: data is encoded, data is written on DNA and stored. Consequently, the data is read from DNA once required. Chemically writing DNA is referred to as DNA synthesis. The polymerization process that controls the helical conformation of synthetic DNA-strands is error prone [15]. Reading synthesized DNA is referred to as DNA sequencing. This corresponds to identifying the order of nucleotides in a DNA-sequence. This process is also known to be error prone, especially for DNA-sequences that contain adjacent repeating bases [6]. Depending on the application, different DNA storage architectures adhere to different synthesis and sequencing methods [16].

2.2 Basic concepts of Coding

We present basic properties of codes using sources of information [17] and [3]. Generated data is encoded over the quaternary alphabet $\{0, 1, 2, 3\}$ that represents the nucleotides as follows:

$$A \leftrightarrow 0,\ T \leftrightarrow 1,\ G \leftrightarrow 2,\ C \leftrightarrow 3. \tag{1}$$

Definition 2.1. We define a **DNA-word** $\underline{\mathbf{x}}$ of **length** n as

$$\underline{\mathbf{x}} = [x_1, x_2, ..., x_n] \tag{2}$$

where $x_i \in \{0, 1, 2, 3\}\ \forall i \in \{1, ..., n\}$.

Definition 2.2. We define the set $\boldsymbol{\mathcal{B}}(n)$ of all DNA-words of length n as

$$\mathcal{B}(n) = \{\underline{\mathbf{x}} : \underline{\mathbf{x}} \text{ is a DNA-word of length } n\} \tag{3}$$

with cardinality

$$B(n) = |\mathcal{B}(n)| = 4^n. \tag{4}$$

Example 2.1. We present an example of the set of all DNA-words of length 2 and observe this set indeed has $4^2 = 16$ elements:

$\mathcal{B}(2) = \{[00], [01], [02], [03], [10], [11], [12], [13], [20], [21], [22], [23], [30], [31], [32], [33]\}$.

Next, in our application described in Section 1.2, the GC-content of the DNA-words is essential. This is referred to as the weight *$w(\underline{x})$* of DNA-words. First consider DNA-word $\underline{\mathbf{x}}$ of length n and define

$$w_i = \begin{cases} 0 & x_i \in \{0, 1\}, \\ 1 & x_i \in \{2, 3\}, \end{cases} \tag{5}$$

$\forall i \in \{1, ..., n\}$.

Definition 2.3. We define the **weight** $w(\underline{\mathbf{x}})$ of a DNA-word $\underline{\mathbf{x}}$ of length n as

$$w(\underline{\mathbf{x}}) = \sum_{i=1}^{n} w_i. \tag{6}$$

Example 2.2. We present two examples $\underline{\mathbf{x}}_1$ and $\underline{\mathbf{x}}_2$ of DNA-words with weight $w(\underline{\mathbf{x}}_j) = 5$ for $j \in \{1, 2\}$:

$\underline{\mathbf{x}}_1 = [3, 0, 0, 1, 3, 3, 0, 1, 2, 3]$
$\underline{\mathbf{x}}_2 = [2, 2, 2, 2, 2]$.

Moreover, as discussed in Section 2.1, the number of adjacent symbols in DNA-words is crucial for the erratic behaviour in the DNA sequencing process.

Definition 2.4. We define the **runlength** $r(\underline{\mathbf{x}})$ of a word $\underline{\mathbf{x}}$ as

$$r(\underline{\mathbf{x}}) = \max\{r : \exists i \text{ such that } x_i = x_{i+1} = ... = x_{i+r-1}\}. \tag{7}$$

Example 2.3. We present two examples $\underline{\mathbf{x}}_1$ and $\underline{\mathbf{x}}_2$ of DNA-words with runlength $r(\underline{\mathbf{x}}_j) = 5$ for $j \in \{1, 2\}$:

$\underline{\mathbf{x}}_1 = [1, 0, 3, 3, 3, 2, 0, 0, 0, 0, 0, 1, 0]$
$\underline{\mathbf{x}}_2 = [0, 0, 1, 1, 1, 2, 2, 2, 2, 3, 3, 3, 3, 3]$.

Now, we can select the DNA-words from the set $\mathcal{B}(n)$ that all adhere to an imposed weight w and maximum runlength r.

Definition 2.5. We define the set $\boldsymbol{\mathcal{B}}_r(n,\ w)$ of all DNA-words $\underline{\mathbf{x}}$ of length n that satisfy weight w and maximum runlength r as

$$\mathcal{B}_r(n, w) = \{\underline{\mathbf{x}} : w(\underline{\mathbf{x}}) = w \wedge r(\underline{\mathbf{x}}) \leq r\} \tag{8}$$

with cardinality $B_r(n, w) = |\mathcal{B}_r(n, w)|$ for which a formula is presented in [3].

Example 2.4. We present

$$\begin{aligned} \mathcal{B}_1(3, 1) = \{&[0, 1, 2], [0, 1, 3], [0, 2, 0], [0, 2, 1], [0, 3, 0], [0, 3, 1], [1, 0, 2], [1, 0, 3], \\ &[1, 2, 0], [1, 2, 1], [1, 3, 0], [1, 3, 1], [2, 0, 1], [2, 1, 0], [3, 0, 1], [3, 1, 0]\} \end{aligned} \tag{9}$$

with cardinality 16.

Example 2.5. We present two examples $\underline{\mathbf{x}}_1$ and $\underline{\mathbf{x}}_2$ of DNA-words in $\mathcal{B}_3(5, 2)$:

$\underline{\mathbf{x}}_1 = [3, 3, 0, 0, 0]$
$\underline{\mathbf{x}}_2 = [0, 1, 2, 3, 0]$.

Note that $r(\underline{\mathbf{x}}_2) = 1$ and observe $1 \leq r$ to conclude $\underline{\mathbf{x}}_2 \in \mathcal{B}_3(5, 2)$.

Example 2.6. We present an example $\underline{\mathbf{x}}_3$ such that $\underline{\mathbf{x}}_3 \in \mathcal{B}(5)$ but $\underline{\mathbf{x}}_3 \notin \mathcal{B}_3(5, 2)$:

$\underline{\mathbf{x}}_3 = [2, 2, 3, 0, 1]$

Note $\underline{\mathbf{x}}_3 \notin \mathcal{B}_3(5, 2)$ since $w(\underline{\mathbf{x}}_3) \neq 2$.

Remark 1. From Example 2.6 we obtain the general result $\mathcal{B}_r(n, w) \subseteq \mathcal{B}(n)$.

Now we have defined and discussed sets in which all DNA-words satisfy several restricting properties such as length n, weight w and maximum runlength r. From this set we can take subsets, referred to as DNA-codes.

Definition 2.6. We define a **DNA-code** $\mathcal{C}$ as a set of DNA-words that all satisfy specified weight w, length n and maximum runlength r.

Note $\mathcal{C} \subseteq \mathcal{B}_r(n, w)$.

2.3 A DNA-d code

In previous Section 2.2, we derived a definition of a DNA-code, that consists of DNA-words satisfying universal properties. These properties are referred to as constraints. We introduce another constraint that concerns the number of entries in which DNA-words differ: the Hamming distance.

Definition 2.7. Suppose $\underline{\mathbf{x}}$ and $\underline{\mathbf{y}}$ are DNA-words. We define the **Hamming distance between DNA-words** as

$$H(\underline{\mathbf{x}}, \underline{\mathbf{y}}) = |\{i : x_i \neq y_i\}|. \tag{10}$$

Example 2.7. We present two examples $\underline{\mathbf{x}}$ and $\underline{\mathbf{y}}$ of DNA-words in $\mathcal{B}_1(6, 3)$ that have Hamming distance 3 from each other:

$\underline{\mathbf{x}} = [0, 2, 3, 1, 0, 3]$,
$\underline{\mathbf{y}} = [1, 2, 1, 3, 0, 3]$.

Definition 2.8. Suppose $\mathcal{C}$ is a DNA-code. We define the **Hamming distance d of $\mathcal{C}$** as

$$d = \min\{H(\underline{\mathbf{x}}, \underline{\mathbf{y}}) : \underline{\mathbf{x}}, \underline{\mathbf{y}} \in \mathcal{C}, \underline{\mathbf{x}} \neq \underline{\mathbf{y}}\}. \tag{11}$$

A DNA-code satisfies minimum Hamming distance d if all DNA-words in the DNA-code differ mutually differ in at least d positions.

Definition 2.9. Suppose $\mathcal{C}$ is a DNA-code. We define a **DNA-d code $\mathcal{C}_d$** as the subset of $\mathcal{C}$ that satisfies a minimum Hamming distance d, hence $H(\underline{\mathbf{x}}, \underline{\mathbf{y}}) \geq d$, $\forall \underline{\mathbf{x}}, \underline{\mathbf{y}} \in \mathcal{C}$, $\underline{\mathbf{x}} \neq \underline{\mathbf{y}}$.

Definition 2.10. We define the **size** $|\mathcal{C}_d|$ of a DNA-d code as the number of DNA-words it contains.

Definition 2.11. We define the **size of the largest DNA-d code** $B_r(n, w, d)$ as

$$B_r(n, w, d) = \max\{|\mathcal{C}_d| : \ \mathcal{C}_d \subseteq \mathcal{B}_r(n, w)\}. \tag{12}$$

Up to now, there has not been found a general formula for the cardinality $B_r(n, w, d)$ of this largest possible DNA-d code. For the case $d = 2$, a formula for $B_1(n, w, 2)$ is presented in [4].

Example 2.8. Building upon Example 2.4, we present a DNA-2 code $\mathcal{C}_2$ such that $\mathcal{C}_2 \subseteq \mathcal{B}_1(3, 1)$

$$\mathcal{C}_2 = \{[0, 1, 2], [0, 2, 0], [0, 3, 1], [1, 0, 2], [1, 2, 1], [1, 3, 0], [2, 0, 1], [2, 1, 0]\}. \tag{13}$$

with cardinality 8.

From [3] and [4], conclude that the maximum size for DNA-2 codes with DNA-words satisfying $n = 3$, $w = 1$, $r = 1$ is 8. Observe for $\mathcal{C}_2$ in Example 2.8 it holds that $|\mathcal{C}_2| = B_1(3, 1, 2)$.

3 The impact of constraints

Imposing constraints to DNA-codes reduces their potential size, as we have seen in Sections 2.2 and 2.3 respectively. This means less information can be stored in the DNA-codes. Yet we adhere to these constraints for valid reasons, presented in this Chapter.

3.1 GC-weight w

A fixed GC-content or weight w that a DNA-code satisfies, determines the erratic behaviour of a DNA Based Storage System to a large extent [6]. The weight of a DNA-word is defined to be the accumulated number of G and C nucleotides it hosts. The DNA-code satisfies the universal weight w if all DNA-words in it have an equal GC-content. As the number of G and C nucleotides is a determining factor for the melting temperature of (synthesized) DNA, it is desirable in DNA Based Storage Systems that all DNA-words in a DNA-code have the same GC-content [18].

Not only is it desirable that a DNA-code satisfies this universal weight, the percentage of G and C nucleotides in DNA-words is preferred to be around half the word length n. If this rate exceeds 60% for example, errors in the laboratory technique that amplifies DNA-sequences occur [19]. We conclude a high weight makes DNA synthesizing error prone. It also leads to erratic behaviour in DNA sequencing methods that are rather robust, such as Low Coverage sequencing.

High GC-content should be avoided and the desired biochemical properties for DNA synthesizing and -reading are met at a rates between 45% and 55% [10]. A DNA-code is said to be balanced if the code words satisfy $w = \lfloor \frac{n}{2} \rfloor$ [13]. Throughout this research we mostly consider balanced codes.

3.2 Maximum runlength r

A DNA-code adheres to a maximum runlength r if the maximum number of adjacent identical symbols, or homopolymer run, in all DNA-words is limited by r. Imposing a maximum runlength constraint to a DNA-code is proven to be beneficial and we will elaborate on these so called Runlength-Limited Sequences [20]. By limiting the repetition of symbols we can avoid the occurrence of destructive patterns. A limited maximum runlength ensures adequate frequency for the synchronization of the DNA-sequencing mechanism. For example, a long run of T's in $AGCTTTTTAGCGC$ increases the loss rate of DNA information as it can be read as a short run of T's [1].

Allowing repetitions by relaxing the runlength constraint gives larger possible DNA-code sizes. Upper- and lower bounds for $B_2(n, w, 2)$ and $B_3(n, w, 2)$ are presented in [5]. Research [21] has indicated changes in error rates in various DNA-sequencing methods at long homopolymer runs in DNA-strands.

Throughout this research we mostly adhere to the no-runlength constraint, corresponding to $r = 1$. This limits the amount of information that can be stored in a DNA-d code, but increasing r makes the DNA-d code more prone to sequencing errors [6].

3.3 Minimum Hamming distance d

From Section 2.3, we observe that adhering to a minimum Hamming distance d limits the number of possible DNA-words a DNA-d can possibly host. But, the minimum Hamming distance of a code is an important parameter of a DNA-code, since we can deduce the error protection capabilities of the DNA-code from it [17]. Both error detection and error correction are discussed throughout this Section.

Recall DNA sequencing is the process of reading DNA-d codes after storage. Data stored as DNA is clearly fragile. Consider a DNA-d code $\mathcal{C}_d$:

$$\mathcal{C}_d = \{\underline{\mathbf{c}}_i : i \in \{1, ..., |\mathcal{C}_d|\}\} \tag{14}$$

In a DNA-Based Storage System, data appears in several forms from source to destination. At the source, the raw data is forwarded as a message. Next, this information is encoded to a DNA-word from $\mathcal{C}_d$. After storage, a DNA-word $\underline{\mathbf{x}}$ is received, possibly in an alternated form due to sequencing or storing errors. Decoding mechanisms are explored to work around these errors. Depending on the purposes of the information after decoding this received DNA-word, the receiver is faced with a choice for a decoding mechanism. We consider both error detection and error correction.
An error detecting mechanism determines whether a received DNA-word corresponds with one of the DNA-words in $\mathcal{C}_d$. Applying a detection mechanism, the receiver can choose to interpolate and only decode the received DNA-words $\underline{\mathbf{x}}$ that correspond with some $\underline{\mathbf{c}}_i \in \mathcal{C}_d$.
An error correction mechanism decodes the received DNA-word $\underline{\mathbf{x}}$ as $\underline{\mathbf{y}}$, corresponding to one of the DNA-words $\underline{\mathbf{c}}_i$ from $\mathcal{C}_d$ with the smallest Hamming distance to $\underline{\mathbf{x}}$:

$$\underline{\mathbf{y}} \in \{\underline{\mathbf{c}}_i : H(\underline{\mathbf{x}}, \underline{\mathbf{c}}_i) = \min\{H(\underline{\mathbf{x}}, \underline{\mathbf{c}}_i)\}, \underline{\mathbf{c}}_i \in \mathcal{C}_d\} \tag{15}$$

Depending on the imposed distance of the DNA-d code, a number of errors can surely be detected or corrected. We introduce a binary compass to illustrate error detection and error correction. Although in DNA Data Storage Systems we have more symbols at our disposal, binary codes suffice to clarify the concepts.

Table 1: A binary compass, satisfying $n = 5$, $d = 3$

Direction	Code
North	00000
East	01011
South	11110
West	10101

Note Table 1 can be interpreted as a DNA-3 code satisfying DNA-word length 5, weight $w = 0$, maximum runlength $r = 5$:

$$\mathcal{C}_3 = \{[0,0,0,0,0], [0,1,0,1,1], [1,1,1,1,0], [1,0,1,0,1]\}. \tag{16}$$

3.3.1 Error detection and pitfalls

Consider Table 1 and suppose the direction West is forwarded, hence $\underline{\mathbf{c}}_4 = [1,0,1,0,1]$ is encoded and the received word is $\underline{\mathbf{x}} = [1,0,0,0,0]$. If an error detection mechanism is applied, it is detected that $\underline{\mathbf{x}}$ does not match one of the encoded directions in Table 1. Theoretically, it is possible $\underline{\mathbf{c}}_3$ was encoded and three errors occurred: in every position except the first and last one. The mechanism would still detect $\underline{\mathbf{x}}$ as erratic. As long as errors are detected, the mechanism prevents $\underline{\mathbf{x}}$ from being decoded.
We consider another case where three errors occur, namely in all symbols except the second and fourth one: the encoded $\underline{\mathbf{c}}_4$ is received as $\underline{\mathbf{x}} = [0,0,0,0,0]$. No errors would be detected as $\underline{\mathbf{x}}$ corresponds with $\underline{\mathbf{c}}_1 \in \mathcal{C}_3$. The detection mechanism failed in this case. We observe we can trust the mechanism without hesitation up to two sequencing errors. To generalize this result, we refer to a result in code theory [22]:

Theorem 1. *In a DNA-d code, up to $d - 1$ sequencing errors are guaranteed to be detected.*

3.3.2 Error correction and pitfalls

We present a similar example as in Chapter 3.3.1 as well as an intuitive result to elaborate on error correction mechanisms.
Consider Table 1 and suppose the direction North is forwarded, hence $\underline{\mathbf{c}}_1 = [0,0,0,0,0]$ is encoded and the received word is $\underline{\mathbf{x}} = [0,1,0,0,0]$. An error correction mechanism would decode this as $\underline{\mathbf{y}} = [0,0,0,0,0]$ according to equation 15 and correct the error. Theoretically, it is possible $\underline{\mathbf{c}}_2$ was encoded and two errors occurred: in the fourth and fifth symbol. The mechanism would fail in this case as it would still decode $\underline{\mathbf{y}} = [0,0,0,0,0]$ according to equation 15. This points out that applying an error correction mechanism in decoding involves risks as the decoded $\underline{\mathbf{y}}$ from the received DNA-word $\underline{\mathbf{x}}$ does not always correspond to encoded $\underline{\mathbf{c}}_i \in \mathcal{C}_d$. To generalize this result, we refer to a result in code theory [22]:

Theorem 2. *In a DNA-d code with $d > 2$, up to $\frac{d-1}{2}$ sequencing errors are guaranteed to be corrected.*

To elaborate on this result, we consider an intuitive example. Imagine a person would live in house A in a village $\mathcal{C}_7$. We obtain that all houses in the village are at least 7 miles apart. The inhabitant starts walking from home in stages of one mile, or, as the metaphor would imply, deviating from the encoded DNA-word A. An error has occurred. Up to 3 miles away, A is still the closest house, no matter in what direction he/she deviates: 3 errors are guaranteed to be corrected.

From Theorems 1 and 2 respectively, we conclude the accuracy level of data transfer through error detection or -correction mechanisms depend on the imposed minimum Hamming distance d of the DNA-d code. This constraint makes decoding DNA-words through error detection and -correction mechanisms more reliable. But in order to satisfy a higher distance, more symbols in each DNA-word are required to encode the same messages, which impacts the information density of DNA-d codes.

We conclude working with DNA-Based Storage System involves finding a balance between efficient DNA-d code generation and the level of accuracy of your decoding mechanisms.

4 The maximum size of a DNA-3 code

In order to optimize the usage of DNA-Based Storage Systems, we aspire to create DNA-codes that satisfy high distances for accuracy and host as much DNA-words as possible for a high information density. As discussed in Section 2.3, a formula for $B_r(n, w, d)$ has not yet been found. We estimate the size of the largest DNA-d code by determining lower bounds counting the DNA-words in a valid DNA-d code. Previous research [4], [3] derived a formula for $B_1(n, w, 2)$. We improve the distance and present an analytical proof on the maximum size of a DNA-3 code satisfying $n = 4$, $w = 2$ and $r = 1$.

Theorem 3. *$B_1(4, 2, 3) = 12$.*

Proof. Throughout this proof DNA-d codes are presented in matrix format in which the rows represent the DNA-words. The elements $c_{i,j}$ of the matrices are displayed as follows: $c_{i,j} : i \in \{1, ..|\mathcal{C}_d|\}, j \in \{1, ..., 4\}$.

Consider a valid DNA-3 code $\mathcal{C}_3 \subseteq \mathcal{B}_1(4, 2)$,

Table 2: $\mathcal{C}_3 \subseteq \mathcal{B}_1(4, 2)$

0	1	2	3
0	2	3	1
0	3	1	2
1	0	3	2
1	2	0	3
1	3	2	0
2	0	1	3
2	1	3	0
2	3	0	1
3	0	2	1
3	1	0	2
3	2	1	0

From Table 2, we conclude the size of $\mathcal{C}_3$ is 12. This is a lower bound for the size of the largest DNA-3 code in $\mathcal{B}_1(4, 2)$, so $B_1(4, 2, 3) \geq 12$.

In Table 2, we obtain 3 DNA-words start with the symbol 0. We suppose there are 4 DNA-words satisfying the same constraints. We obtain:

Table 3: A potential DNA-3 code

0	1	...	...
0	2	...	...
0	3	...	...
0	$c_{4,2}$	...	...

In case $c_{4,2} = 2$, $c_{4,2} = 3$ or $c_{4,2} = 1$, $d = 3$ would no longer be satisfied. In case $c_{4,2} = 0$, the no-runlength constraint would no longer be satisfied. Hence not for any symbol $c_{4,2} \in \{0, 1, 2, 3\}$, the DNA-$d$ code in Table 3 would be a DNA-3 code in $\mathcal{B}_1(4, 2)$. Interchanging rows in the format of Table 3 implies the same result. Choosing an element from $\{1, 2, 3\}$ as a first symbol of all the DNA-words in the format of Table 3 also gives an upper bound of 3 valid DNA-words. Hence $B_1(4, 2, 3) \leq 12$. $\square$

5 The maximum sizes of DNA-n codes

In the search for the largest DNA-d code satisfying a minimum Hamming distance $d > 2$, we consider an extreme constraint, that forces a DNA-d code to satisfy a minimum Hamming distance equal to the length n of the DNA-words: $d = n$. We will determine $B_r(n, w, n)$ in this Chapter. Before we do so, we introduce a concept that is used throughout the proofs in this Chapter.

Definition 5.1. We define **cascading** as placing matrices after each other, so that we obtain a new, larger matrix.

We illustrate the concept with an example.

Example 5.1. Define $A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$ and $B = \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}$.
Cascading A a number of 3 times gives a new matrix $\mathcal{A}$ defined as

$$\mathcal{A} = \begin{bmatrix} 1 & 2 & 1 & 2 & 1 & 2 \\ 3 & 4 & 3 & 4 & 3 & 4 \end{bmatrix}. \tag{17}$$

Cascading A with B that is cascaded a number of 2 times gives a new matrix $\mathcal{A}_2$ defined as

$$\mathcal{A}_2 = \begin{bmatrix} 1 & 2 & 0 & 0 & 0 & 0 \\ 3 & 4 & 0 & 0 & 0 & 0 \end{bmatrix}. \tag{18}$$

Throughout this Chapter, we present DNA-n codes in matrix format in which the rows represent the DNA-words. The elements $c_{i,j}$ of the matrices are displayed as follows: $c_{i,j} : i \in \{1, ...|\mathcal{C}_n|\}, j \in \{1, ..., n\}$.

We obtain that each element $c_{i,j} \in \{0, 1, 2, 3\}$ can appear at most once in every column. Consider a potential DNA-n code:

Table 4: A potential DNA-n code

0	...	$c_{1,n}$
1	...	$c_{2,n}$
2	...	$c_{3,n}$
3	...	$c_{4,n}$
$c_{5,1}$	....	$c_{5,n}$

We define the number of rows or the size of the DNA-n code as $s = |\mathcal{C}_n|$. From equation (10), we observe the constraint $d = n$ requires

$$c_{1,j} \neq c_{2,j} \neq ... \neq c_{s,j} \; \forall j \in \{1, ..., n\}. \tag{19}$$

Implement any element $c_{5,1} \in \{0, 1, 2, 3\}$ in Table 4 and observe $\exists i \in \{1, 2, 3, 4\} : c_{5,1} = c_{i,1}$. This contradicts equation (19) for $j = 1$. We observe $d = n$ can not be satisfied and conclude $s \leq 4$ for each valid $\mathcal{C}_n$. We observe that allowing repetition of adjacent symbols in DNA-words by setting $r > 1$ or assigning a specific weight w to the DNA-n code would not improve the maximum size of $\mathcal{C}_n$. In other words:

$$B_r(n, w, n) \leq 4 \; \forall n, w, r. \tag{20}$$

Can we create a valid DNA-n code with 4 DNA-words satisfying all imposed constraints and conclude $B_r(n, w, n) \geq 4$? We observe from equation (8) that DNA-n codes satisfying $r = 1$ satisfy all $r > 1$ as well. We will determine $B_r(n, w, n)$ for all possible weights w.

We first consider even lengths n and define $w = \frac{n}{2}$, as this is considered to be beneficial for the accuracy of the DNA-Based Storage System in Section 3.1.

Theorem 4. *For n an even number, $B_r(n, \frac{n}{2}, n) = 4 \; \forall r \geq 1$.*

Proof. Define a building block for a DNA-n code, displayed in matrix format:

Table 5: Building block $\mathcal{C}_2 \subseteq \mathcal{B}_1(2,1)$ for a DNA-n code

0	2
1	3
2	0
3	1

For all even n, cascading the building block from Table 5 a number of $\frac{n}{2}$ times represents a valid code DNA-n code $\mathcal{C}_n \subseteq \mathcal{B}_1(n, \frac{n}{2})$. We conclude $B_r(n, \frac{n}{2}, n) \geq 4 \; \forall r \geq 1$ and in combination with equation (20) this concludes the proof. □

Theorem 5. *For $\lceil \frac{n}{3} \rceil \leq w \leq \lfloor \frac{2n}{3} \rfloor$, $w \neq \frac{n}{2}$, $B_r(n, w, n) = 3 \; \forall r \geq 1$.*

Proof. Define 3 building blocks for a DNA-n code, displayed in matrix format:

Table 6: Building block $\mathcal{C}_2 \subseteq \mathcal{B}_1(2,1)$ for a DNA-n code

0	3
1	2
2	1

Table 7: Building block $\mathcal{C}_3 \subseteq \mathcal{B}_1(3,1)$ for a DNA-n code

0	2	1
1	0	3
2	1	0

Table 8: Building block $\mathcal{C}_3 \subseteq \mathcal{B}_1(3,2)$ for a DNA-n code

2	0	3
0	3	2
3	2	1

Let $a \in \mathbb{N}$ satisfy $0 \leq a < \lceil \frac{n}{3} \rceil$ and let $w^\star = \lceil \frac{n}{3} \rceil + a$, $w^\star \neq \frac{n}{2}$. We consider all possible lengths n of the DNA-words.

- $n = 0 \bmod 3$, hence $\lceil \frac{n}{3} \rceil = \frac{n}{3}$.

  Cascading the building block $\mathcal{C}_3 \subseteq \mathcal{B}_1(3,1)$ from Table 7 a number of $\frac{2n}{3} - w^\star = \frac{n}{3} - a$ times gives a valid DNA-$(n-3a)$ code:

$$\mathcal{C}_{n-3a} \subseteq \mathcal{B}_1\left(n - 3a, \frac{n}{3} + a\right). \tag{21}$$

  Cascading the obtained $\mathcal{C}_{n-3a}$ with the building block $\mathcal{C}_3 \subseteq \mathcal{B}_1(3,2)$ from Table 8 that is cascaded a number of $w^\star - \frac{n}{3} = a$ times gives a valid DNA-n code:

$$\mathcal{C}_n \subseteq \mathcal{B}_1\left(n, \frac{n}{3} + a\right) = \mathcal{B}_1\left(n, w^\star\right). \tag{22}$$

- $n = 1 \bmod 3$, hence $\lceil \frac{n}{3} \rceil = \frac{n-1}{3} + 1$.

  Cascading the building block $\mathcal{C}_2 \subseteq \mathcal{B}_1(2,1)$ from Table 6 a number of 2 times gives a valid DNA-4 code:

$$\mathcal{C}_4 \subseteq \mathcal{B}_1\left(4, 2\right). \tag{23}$$

  Cascading the obtained $\mathcal{C}_4$ with the building block $\mathcal{C}_3 \subseteq \mathcal{B}_1(3,1)$ from Table 7 that is cascaded a number of $\frac{2n-2}{3} - w^\star = \frac{n-1}{3} - 1 - a$ times gives a valid DNA-$(n-3a)$ code:

$$\mathcal{C}_{n-3a} \subseteq \mathcal{B}_1\left(n - 3a, \frac{(n-1)}{3} + 1 - a\right). \tag{24}$$

  Cascading the obtained $\mathcal{C}_{n-3a}$ with the building block $\mathcal{C}_3 \subseteq \mathcal{B}_1(3,2)$ from Table 8 that is cascaded a number of $w^\star - \frac{n+2}{3} = a$ times gives a valid DNA-n code:

$$\mathcal{C}_n \subseteq \mathcal{B}_1\left(n, \frac{n-1}{3} + 1 + a\right) = \mathcal{B}_1\left(n, w^\star\right). \tag{25}$$

- $n = 2 \bmod 3$, hence $\lceil \frac{n}{3} \rceil = \frac{n-2}{3} + 1$.

  Cascading the building block $\mathcal{C}_2 \subseteq \mathcal{B}_1(2,1)$ from Table 6 with the building block $\mathcal{C}_3 \subseteq \mathcal{B}_1(3,1)$ from Table 7 that is cascaded a number of $\frac{2n-1}{3} - w^\star = \frac{n+1}{3} - 1 - a$ times gives a valid DNA-$(n-3a)$ code:

$$\mathcal{C}_{n-3a} \subseteq \mathcal{B}_1\left(n - 3a, \frac{n+1}{3} - a\right). \tag{26}$$

  Cascading the obtained $\mathcal{C}_{n-3a}$ with the building block $\mathcal{C}_3 \subseteq \mathcal{B}_1(3,2)$ from Table 8 that is cascaded a number of $w^\star - \frac{n+1}{3} = a$ times gives a valid DNA-n code:

$$\mathcal{C}_n \subseteq \mathcal{B}_1\left(n, \frac{n+1}{3} + a\right) = \mathcal{B}_1\left(n, \frac{n-2}{3} + 1 + a\right) = \mathcal{B}_1(n, w^\star). \tag{27}$$

We obtain DNA-n codes in equations (22) , (25) and (27) respectively. For every possible DNA-word length n, we observe that for all $a \in \mathbb{N}$ satisfying $0 \leq a < \lceil \frac{n}{3} \rceil$ hence for all $w^\star \in \{\lceil \frac{n}{3} \rceil, ..., \lfloor \frac{2n}{3} \rfloor\}$, $w^\star \neq \frac{n}{2}$, we can find a cascading scheme such that we can obtain DNA-n codes of size 3. We conclude $B_1(n, w^\star, n) \geq 3 \; \forall r \geq 1$.

Suppose we can create a DNA-n code $\mathcal{C}_n$ such that $|\mathcal{C}_n| = B_1(n, w^\star, n) = 4$. From (the text before) equation (19) we obtain each column of $\mathcal{C}_n$ in matrix format has exactly 2 elements from $\{2,3\}$. This implies $\mathcal{C}_n$ contains exactly $2n$ elements from $\{2,3\}$. We know each row of $\mathcal{C}_n$ in matrix format has $w^\star$ elements from $\{2,3\}$ by definition of the weight $w^\star$. Hence $\mathcal{C}_n$ contains exactly $4w^\star$ elements from $\{2,3\}$ in total. These observations yield $2n = 4w^\star$, so $w^\star = \frac{n}{2}$. This contradicts the assumption $w^\star \neq \frac{n}{2}$ and we conclude $B_r(n, w^\star, n) \leq 3 \; \forall r \geq 1$. □

Theorem 6. *For $0 \leq w < \lceil \frac{n}{3} \rceil$ or $\lfloor \frac{2n}{3} \rfloor < w \leq n$, $B_r(n, w, n) = 2$.*

Proof. Define 5 building blocks for a DNA-n code, displayed in matrix format:

Table 9: Building block $\mathcal{C}_1 \subseteq \mathcal{B}_1(1, 0)$ for a DNA-n code

0
1

Table 10: Building block $\mathcal{C}_1 \subseteq \mathcal{B}_1(1, 1)$ for a DNA-n code

3
2

Table 11: Building block $\mathcal{C}_2 \subseteq \mathcal{B}_1(2, 0)$ for a DNA-n code

1	0
0	1

Table 12: Building block $\mathcal{C}_2 \subseteq \mathcal{B}_1(2, 2)$ for a DNA-n code

2	3
3	2

Table 13: Building block $\mathcal{C}_2 \subseteq \mathcal{B}_1(2, 1)$ for a DNA-n code

1	2
3	0

Let $w^\star \in \mathbb{N}$ satisfy $0 \leq w^\star < \lceil \frac{n}{3} \rceil$. We consider all possible lengths n of the DNA-words.

- n even.

  Cascading the building block $\mathcal{C}_2 \subseteq \mathcal{B}_1(2, 0)$ from Table 11 a number of $\frac{n}{2} - w^\star$ times gives a valid DNA-$(n - 2w^\star)$ code:
  $$\mathcal{C}_{n-2w^\star} \subseteq \mathcal{B}_1(n - 2w^\star, 0). \tag{28}$$

- n odd, hence $\lfloor \frac{n}{2} \rfloor = \frac{n+1}{2} - 1$.

  Cascading the building block $\mathcal{C}_1 \subseteq \mathcal{B}_1(1, 0)$ from Table 9 with the building block $\mathcal{C}_2 \subseteq \mathcal{B}_1(2, 0)$ from Table 11 that is cascaded a number of $\lfloor \frac{n}{2} \rfloor - w^\star$ times gives a valid DNA-$(n - 2w^\star)$ code:
  $$\mathcal{C}_{n-2w^\star} \subseteq \mathcal{B}_1(n - 2w^\star, 0). \tag{29}$$

Cascading the obtained $\mathcal{C}_{n-2w^\star}$ in equations (28) and (29) separately with the building block $\mathcal{C}_2 \subseteq \mathcal{B}_1(2, 1)$ from Table 13 that is cascaded a number of $w^\star$ times gives valid DNA-n codes:
$$\mathcal{C}_n \subseteq \mathcal{B}_1(n, w^\star). \tag{30}$$

Let $\underline{w} = n - w^\star$ and observe $\lfloor \frac{2n}{3} \rfloor < \underline{w} \leq n$.

- n even.

  Cascading the building block $\mathcal{C}_2 \subseteq \mathcal{B}_1(2,2)$ from Table 12 a number of $\underline{w} - \frac{n}{2}$ times gives a valid DNA-$(2\underline{w} - 2)$ code:
  $$\mathcal{C}_{2\underline{w}-n} \subseteq \mathcal{B}_1(2\underline{w} - n, 2\underline{w} - n). \tag{31}$$

- n is odd, hence $\lceil \frac{n}{2} \rceil = \frac{n+1}{2}$.

  Cascading the building block $\mathcal{C}_1 \subseteq \mathcal{B}_1(1,1)$ from Table 10 with the building block $\mathcal{C}_2 \subseteq \mathcal{B}_1(2,2)$ from Table 12 that is cascaded a number of $\underline{w} - \lceil \frac{n}{2} \rceil$ times gives a valid DNA-$(2\underline{w} - n)$ code:
  $$\mathcal{C}_{2\underline{w}-n} \subseteq \mathcal{B}_1(2\underline{w} - n, 2\underline{w} - n). \tag{32}$$

Cascading the obtained $\mathcal{C}_{2\underline{w}-n}$ in equations (31) and (32) separately with the building block $\mathcal{C}_2 \subseteq \mathcal{B}_1(2,1)$ from Table 13 that is cascaded a number of $n - \underline{w}$ times gives valid DNA-n codes:
$$\mathcal{C}_n \subseteq \mathcal{B}_1(n, \underline{w}). \tag{33}$$

We obtain two DNA-n codes represented in equation (30) and two DNA-n codes represented in equation (33). For every possible DNA-word length n, we observe that for all $w^\star$ such that $0 \leq w^\star < \lceil \frac{n}{3} \rceil$ and for all $\underline{w}$ such that $\lfloor \frac{2n}{3} \rfloor < \underline{w} \leq n$, we can find a cascading scheme such that we obtain DNA-n codes of size 2. We conclude $B_1(n, w, n) \geq 2\ \forall r \geq 1, w = w^\star$ or $w = \underline{w}$.

Suppose we can create a DNA-n code C_n such that $|C_n| = B_1(n, w, n) = 3$ for $w = w^\star$ or $w = \underline{w}$. From (the text before) equation (19) we obtain each column of $\mathcal{C}_n$ in matrix format has exactly one or exactly two elements from $\{2,3\}$. This implies $\mathcal{C}_n$ contains a minimum of n elements from $\{2,3\}$ and a maximum of $2n$ elements from $\{2,3\}$. We know each row of $\mathcal{C}_n$ in matrix format has w elements from $\{2,3\}$ by definition of the weight w. Hence $\mathcal{C}_n$ contains exactly $3w$ elements from $\{2,3\}$ in total. These observations yield $n \leq 3w \leq 2n$, so $\lceil \frac{n}{3} \rceil \leq w \leq \lfloor \frac{2n}{3} \rfloor$. This constradicts the assumption $w = w^\star$ or $w = \underline{w}$ and we conclude $B_r(n, w, n) \leq 2\ \forall r \geq 1, w = w^\star$ or $w = \underline{w}$.

□

In conclusion, theorems 4, 5 and 6 show:
$$\forall r, n \geq 1,\ B_r(n, w, n) = \begin{cases} 4 & w = \frac{n}{2} \text{ for } n \text{ even},\ n \neq \frac{n}{2}, \\ 3 & w \in \{\lceil \frac{n}{3} \rceil, ..., \lfloor \frac{2n}{3} \rfloor\}, w \neq \frac{n}{2}, \\ 2 & 0 \leq w < \lceil \frac{n}{3} \rceil \text{ or } \lfloor \frac{2n}{3} \rfloor < w \leq n. \end{cases} \tag{34}$$

It is obtained that the closer to $\frac{n}{2}$ the imposed weight w of a DNA-d code is restricted, the more DNA-words the DNA-d code can theoretically host. So a balanced DNA-d code is not only desirable for thermal and biochemical purposes as stated in Section 3.1, it is also beneficial for the information storage density of a DNA Based Storage System.

6 Algorithms generating DNA-d codes and improving existing DNA-d code sizes

Note:

- *The results of the algorithms that will be explained are presented in Table 14 and Table 15.*
- *The Pyhthon codes of the algorithms are attached in Appendix A*

In Chapters 4 and 5, we presented analytical derivations of the maximum sizes of DNA-d codes satisfying $d = 3$ and $d = n$ respectively. We consider cases where $d > 2$. In the search for the largest possible DNA-d codes satisfying a specific minimum Hamming distance d among other imposed constrained, we design algorithms that generate valid DNA-d codes and return the size of these codes. Throughout this Chapter we impose the no-runlength constraint $r = 1$ and $w = \lfloor \frac{n}{2} \rfloor$. We obtain lower bounds for $B_1(n, \lfloor \frac{n}{2} \rfloor, d)$ considering the cases $d = 3$ and $d = 4$. We clarify recurring terms first.

Definition 6.1. We define the j**-Neighbourhood** $N_j(\underline{\mathbf{x}})$ or **set of j-Neighbours of a DNA-word** $\underline{\mathbf{x}}$ as:

$$N_j(\underline{\mathbf{x}}) = \{\underline{\mathbf{y}} : H(\underline{\mathbf{x}}, \underline{\mathbf{y}}) \leq j,\ \underline{\mathbf{x}} \neq \underline{\mathbf{y}}\}. \tag{35}$$

Where $H(\underline{\mathbf{x}}, \underline{\mathbf{y}})$ is the Hamming distance between DNA-words as defined in definition 2.7. We denote the size of the j-Neighbourhood as $|N_j(\underline{\mathbf{x}})|$.

Definition 6.2. For DNA-words $\underline{\mathbf{x}}$ and $\underline{\mathbf{y}}$ of length n, $\underline{\mathbf{x}}$ is of higher **lexicographical order** than $\underline{\mathbf{y}}$ if for $\min\{i \in \{1, ..., n\} : x_i \neq y_i\}$ it holds that $x_i > y_i$.

We present two examples to clarify the concept.

Example 6.1. Consider the DNA-3 code $\mathcal{C}_3 = \{[0, 0, 3, 1, 2], [0, 1, 0, 3, 2]\} \subseteq \mathcal{B}_2(5, 2)$. Presented like this, the DNA-words in $\mathcal{C}_3$ are lexicographically ordered as $\underline{\mathbf{c}}_1$ is of lower lexicographical order than $\underline{\mathbf{c}}_2$.

Definition 6.3. We define $\mathcal{D}^\star$ as the set of keys of a dictionary $\mathcal{D}$.

Example 6.2. Consider a lexicographically ordered dictionary:

$$\mathcal{D} = \{[0, 1, 2] : [[0, 1, 3]], [1, 0, 3] : [[1, 0, 2]], [1, 2, 0] : [[0, 2, 0], [1, 2, 1], [1, 3, 0]]\}. \tag{36}$$

It is concluded that the lexicographical order is restricted by the order of the keys of $\mathcal{D}$. The size of $\mathcal{D}$ is determined by the number of keys and denoted as $|\mathcal{D}| = |\mathcal{D}^\star| = 3$.

6.1 Reference algorithms

We present and analyze three reference Algorithms from [4]. The first algorithm that we present creates a DNA-d code through deleting DNA-words $\underline{\mathbf{x}} \in \mathcal{D}^\star$ from a lexicographically ordered dictionary $\mathcal{D}$ in which $\underline{\mathbf{x}}$ is a key and $N_{d-1}(\underline{\mathbf{x}})$ is its value. The second algorithm that we present creates a DNA-d code $\mathcal{C}^\star$ through selecting DNA-words $\underline{\mathbf{x}} \in \mathcal{D}^\star$ from a lexicographically ordered dictionary $\mathcal{D}$ in which $\underline{\mathbf{x}}$ is a key and $N_{d-1}(\underline{\mathbf{x}})$ is its value. The third algorithm we present creates a DNA-d code $\mathcal{C}^\star$ by picking DNA-words $\underline{\mathbf{y}}$ from a lexicographically ordered DNA-code $\mathcal{C}$ that satisfy $H(\underline{\mathbf{x}}, \underline{\mathbf{y}}) \geq d$. Before the first iteration, $\mathcal{C} = \mathcal{B}_1(n, \lfloor \frac{n}{2} \rfloor)$.

6.1.1 Reference Algorithm 1

In the first algorithm we present, in each iteration, one of the DNA-words with the most $d-1$-Neighbours is deleted from a lexicographically ordered dictionary $\mathcal{D}$. In other words: the DNA-word $\underline{\mathbf{x}} \in \mathcal{D}^\star$ of lowest lexicographical order in $\mathcal{D}$ satisfying $|N_{d-1}(\underline{\mathbf{x}})| = \max\{|N_{d-1}(\underline{\mathbf{x}})| : \underline{\mathbf{x}} \in \mathcal{D}^\star\}$ is deleted from $\mathcal{D}$. Before the first iteration, $\mathcal{D} = \{\{\underline{\mathbf{x}} : N_{d-1}(\underline{\mathbf{x}})\} : \underline{\mathbf{x}} \in \mathcal{B}_1(n, \lfloor\frac{n}{2}\rfloor)\}$. The algorithm iterates until all DNA-words $\underline{\mathbf{x}}$ that are keys of $\mathcal{D}$ have no more $d-1$-Neighbours: $|N_{d-1}(\underline{\mathbf{x}}_i)| = 0 \ \forall i \in \{1, ..., |\mathcal{D}|\}$. Then the set of keys of $\mathcal{D}$ represents a valid DNA-d code generated by the algorithm. The algorithm returns $|\mathcal{D}|$: a lower bound for the maximum size $B_1(n, \lfloor\frac{n}{2}\rfloor, d)$.

Algorithm 1 Creating a DNA-d code by deleting DNA-words with the most $d-1$-Neighbours. Returning a lower bound for $B_1(n, \lfloor\frac{n}{2}\rfloor, d)$.

Input: Cosntraints:

- maximum runlength $r = 1$;
- DNA-word length n;
- weight $w = \lfloor\frac{n}{2}\rfloor$;
- minimum Hamming distance d.

Output: A lower bound for $B_1(n, \lfloor\frac{n}{2}\rfloor, d)$.
Data:

- A lexicographically ordered dictionary *words_in_sphere*: $\mathcal{D}_1 = \{\{\underline{\mathbf{x}}_i : N_{d-1}(\underline{\mathbf{x}}_i)\} : i \in \{1, ..., B_1(n, \lfloor\frac{n}{2}\rfloor)\}\}$;
- A lexicographically ordered dictionary *distances*: $\mathcal{D}_2 = \{\{\underline{\mathbf{x}}_i : |N_{d-1}(\underline{\mathbf{x}}_i)|\} : i \in \{1, ..., B_1(n, \lfloor\frac{n}{2}\rfloor)\}\}$.

```
Function Iteration(distances, words_in_sphere):
    word = first key with maximum value in distances $\mathcal{D}_2$;        ▷ See comment 1 below
    word is deleted from both dictionaries $\mathcal{D}_1$ and $\mathcal{D}_2$;
    Both dictionaries $\mathcal{D}_1$ and $\mathcal{D}_2$ are updated;        ▷ See comment 2 below
    return words_in _sphere $\mathcal{D}_1$, distances $\mathcal{D}_2$.

Function Algorithm:
    While the maximum value in distances is greater than 0: Iteration;
    return $|\mathcal{D}_1|$: a lower bound for $B_1(n, \lfloor\frac{n}{2}\rfloor, d)$.        ▷ See comment 3 below
```

Comments

1. The algorithm automatically selects the DNA-word $\underline{\mathbf{x}}$ of lowest lexicographical order in $\mathcal{D}_2$ satisfying $|N_{d-1}(\underline{\mathbf{x}})| = \max\{|N_{d-1}(\underline{\mathbf{x}})| : \underline{\mathbf{x}} \in \mathcal{D}_2^\star\}$. This is discussed in detail in Section 6.3.
2. If *word* was a $d-1$ Neighbour of one of the keys that is still left in the dictionaries, the value of this key should be subtracted.
3. In the Python code in Appendix A this is referred to as the length of *words_in _sphere*.

6.1.2 Reference Algorithm 2

In the second algorithm, an empty list $\mathcal{C}$ is created, which will eventually represent a valid DNA-d code. In each iteration, a DNA-word with the least $d-1$ Neighbours is appended to this list. In other words: the DNA-word $\underline{\mathbf{x}} \in \mathcal{D}^\star$ of lowest lexicographical order in $\mathcal{D}$ satisfying $|N_{d-1}(\underline{\mathbf{x}})| = \min\{(|N_{d-1}(\underline{\mathbf{x}})| : \underline{\mathbf{x}} \in \mathcal{D}^\star\}$ is appended to $\mathcal{C}$ and deleted from $\mathcal{D}$. Afterwards, $\underline{\mathbf{x}}$ and the set $\{\underline{\mathbf{y}} : \underline{\mathbf{y}} \in N_{d-1}(\underline{\mathbf{x}})\}$ are deleted from $\mathcal{D}$ such that these cannot end up in $\mathcal{C}$. The algorithm iterates until $|\mathcal{D}| = 0$. Then $\mathcal{C}$ represents a valid DNA-d code generated by the algorithm. The algorithm returns $|\mathcal{C}|$: a lower bound for the maximum size $B_1(n, \lfloor \frac{n}{2} \rfloor, d)$.

Algorithm 2 Creating a DNA-d code by selecting DNA-words with the least $d-1$-Neighbours. Returning a lower bound for $B_1(n, \lfloor \frac{n}{2} \rfloor, d)$.

Input: Cosntraints:

- maximum runlength $r = 1$;
- DNA-word length n;
- weight $w = \lfloor \frac{n}{2} \rfloor$;
- minimum Hamming distance d.

Output: A lower bound for $B_1(n, \lfloor \frac{n}{2} \rfloor, d)$.
Data:

- A lexicographically ordered dictionary *words_in_sphere* $\mathcal{D}_1 = \{\{\underline{\mathbf{x}}_i : N_{d-1}(\underline{\mathbf{x}}_i)\} : i \in \{1, ..., B_1(n, \lfloor \frac{n}{2} \rfloor)\}\}$;
- A lexicographically ordered dictionary *distances* $\mathcal{D}_2 = \{\{\underline{\mathbf{x}}_i : |N_{d-1}(\underline{\mathbf{x}}_i)|\} : i \in \{1, ..., B_1(n, \lfloor \frac{n}{2} \rfloor)\}\}$;
- An empty list *newcode* $\mathcal{C}$. ▷ See comment 1 below

```
6  Function Iteration(distances, words_in_sphere, newcode):
7      word = first key with minimum value in distances $\mathcal{D}_2$;          ▷ See comment 2 below
       word is appended to newcode $\mathcal{C}$;
       word and the set $\{\underline{word} : \underline{word} \in N(word)\}$ are deleted from dictionaries $\mathcal{D}_1$ and $\mathcal{D}_2$;
       Both dictionaries $\mathcal{D}_1$ and $\mathcal{D}_2$ are updated;            ▷ See comment 3 below
       return distances $\mathcal{D}_2$, words_in_sphere $\mathcal{D}_1$, newcode $\mathcal{C}$.
8
9  Function Algorithm:
10     While $|\mathcal{D}_1| > 0$: Iteration;
11     return $|\mathcal{C}|$: a lower bound for $B_1(n, \lfloor \frac{n}{2} \rfloor, d)$.     ▷ See comment 4 below
```

Comments

1. This will eventually represent the generated DNA-d code.
2. The algorithm automatically selects the DNA-word $\underline{\mathbf{x}}$ of lowest lexicographical order in $\mathcal{D}_2$ satisfying $|N_{d-1}(\underline{\mathbf{x}})| = \min\{(|N_{d-1}(\underline{\mathbf{x}})| : \underline{\mathbf{x}} \in \mathcal{D}_2^\star\}$. This is discussed in detail in Section 6.3.
3. If *word* or *<u>word</u>* was a $d-1$-Neighbour of one of the keys that is still left in the dictionaries, the value of this key should be subtracted.
4. In the Python code in Appendix A this is referred to as the length of *newcode*.

6.1.3 Reference Algorithm 3

We present a third algorithm that creates a DNA-d code $\mathcal{C}^\star$ by selecting DNA-words from a lexicographically ordered DNA-code $\mathcal{C}$. Before the first iteration, $\mathcal{C} = \mathcal{B}_1(n, \lfloor \frac{n}{2} \rfloor)$. In each iteration, the DNA-word $\underline{\mathbf{y}}$ from $\{\underline{\mathbf{y}} \in \mathcal{C} :\ H(\underline{\mathbf{x}}, \underline{\mathbf{y}}) \geq d\ \forall \underline{\mathbf{x}} \in \mathcal{C}^\star\}$ of lowest lexicographical order in $\mathcal{C}$, will be appended to $\mathcal{C}^\star$ and deleted from $\mathcal{C}$. The algorithm iterates until $|\mathcal{C}| = 0$. Then $\mathcal{C}^\star$ represents a valid DNA-d code generated by the algorithm. The algorithm returns $|\mathcal{C}^\star|$: a lower bound for the maximum size $B_1(n, \lfloor \frac{n}{2} \rfloor, d)$.

Algorithm 3 Creating a DNA-d code $\mathcal{C}^\star$ by selecting DNA-words from DNA-code $\mathcal{C}$ such that the minimum Hamming distance of $\mathcal{C}^\star$ is maintained.

```
Input: Cosntraints:
    • maximum runlength $r = 1$;
    • DNA-word length $n$;
    • weight $w = \lfloor \frac{n}{2} \rfloor$;
    • minimum Hamming distance $d$.
Output: A lower bound for $B_1(n, \lfloor \frac{n}{2} \rfloor, d)$.
Data:
    • A lexicographically ordered DNA-code finalwords: $\mathcal{C} = \mathcal{B}_1(n, \lfloor \frac{n}{2} \rfloor)$;
    • An empty list newcode: $\mathcal{C}^\star$.                ▷ See comment 1 below
12 Function Iteration(finalwords, newcode, d):
13     word = the first element of finalwords $\mathcal{C}$;
       If word satisfies $H(word, \underline{word}) \geq d\ \forall \underline{word} \in$ newcode $\mathcal{C}^\star$: word is appended to newcode $\mathcal{C}^\star$;
       word is deleted from finalwords $\mathcal{C}$;
       return finalwords $\mathcal{C}$, newcode $\mathcal{C}^\star$
14
15 Function Algorithm:
16     While $|\mathcal{C}| > 0$: Iteration;
17     return $|\mathcal{C}^\star|$: a lower bound for $B_1(n, \lfloor \frac{n}{2} \rfloor, d)$      ▷ See comment 2 below
```

Comments

1. This will eventually represent the generated DNA-d code.
2. In the Python code in Appendix A this is referred to as the length of *newcode*.

6.2 Non beneficial approaches

We critically analyze the defining properties of the presented reference Algorithms in Section 6.1 to see whether adaptations could improve the lower bounds for $B_1(n, \lfloor \frac{n}{2} \rfloor, d)$. The reference Algorithm 1, presented in Subsection 6.1.1 deletes the DNA-word $\underline{\mathbf{x}}$ of lowest lexicographical order in a dictionary $\mathcal{D}_2$ that satisfies $|N_{d-1}(\underline{\mathbf{x}})| = \max\{(|N_{d-1}(\underline{\mathbf{x}})| : \underline{\mathbf{x}} \in \mathcal{D}_2^\star\}$ in each iteration. The reference Algorithm 2, presented in Subsection 6.1.2 selects the DNA-word $\underline{\mathbf{x}}$ of lowest lexicographical order in a dictionary $\mathcal{D}_2$ that satisfies $|N_{d-1}(\underline{\mathbf{x}})| = \min\{(|N_{d-1}(\underline{\mathbf{x}})| : \underline{\mathbf{x}} \in \mathcal{D}_2^\star\}$ in each iteration. The dictionary $\mathcal{D}_2$ in both Algorithm 1 and Algorithm 2 theoretically contains multiple keys $\underline{\mathbf{x}} \in \mathcal{D}_2^\star$ that satisfy $|N_{d-1}(\underline{\mathbf{x}})| = \max\{(|N_{d-1}(\underline{\mathbf{x}})| : \underline{\mathbf{x}} \in \mathcal{D}_2^\star\}$ or $|N_{d-1}(\underline{\mathbf{x}})| = \min\{(|N_{d-1}(\underline{\mathbf{x}})| : \underline{\mathbf{x}} \in \mathcal{D}_2^\star\}$ respectively. We present sets that represent these keys.

Definition 6.4. We define the set $\mathbf{S_{max}}$ of keys $\underline{\mathbf{x}}$ of a dictionary $\mathcal{D}$ with the most $d-1$-Neighbours as:

$$S_{\max} = \{\underline{\mathbf{x}} \in \mathcal{D}^\star : |N_{d-1}(\underline{\mathbf{x}})| = \max\{(|N_{d-1}(\underline{\mathbf{x}})|\}\}. \tag{37}$$

Definition 6.5. We define the set $\mathbf{S_{min}}$ of keys $\underline{\mathbf{x}}$ of a dictionary $\mathcal{D}$ with the least $d-1$-Neighbours as:

$$S_{\min} = \{\underline{\mathbf{x}} \in \mathcal{D}^\star : |N_{d-1}(\underline{\mathbf{x}})| = \min\{(|N_{d-1}(\underline{\mathbf{x}})|\}\}. \tag{38}$$

6.2.1 Clustering Neighbourhoods

We observe the set of $d-1$-Neighbours of $\underline{\mathbf{x}} \in \mathcal{D}^\star$ is structured as follows:

$$N_{d-1}(\underline{\mathbf{x}}) = \bigcup_{j=1}^{d-1} \{N_j(\underline{\mathbf{x}})\}, \ \underline{\mathbf{x}} \in \mathcal{D}^\star. \tag{39}$$

The reference Algorithm 1 selects the element $\underline{\mathbf{x}} \in \mathcal{D}^\star$ of lowest lexicographical order from $S_{\max}$ and deletes it from $\mathcal{D}$ in each iteration. This selection of $\underline{\mathbf{x}}$ will be adapted in the first iteration. We compare sizes of the subsets $N_j(\underline{\mathbf{x}}) \subseteq N_{d-1}(\underline{\mathbf{x}})$, structured in equation (39). We examine if it has potential to select keys $\underline{\mathbf{x}} \in \mathcal{D}^\star$ that have large j-Neighbourhoods $N_j(\underline{\mathbf{x}}) \subseteq N_{d-1}(\underline{\mathbf{x}})$ for some $j \in \{1, ..., (d-1)\}$ instead of selecting $\underline{\mathbf{x}} \in \mathcal{D}^\star$ from S_{max}, that has a large $|N_{d-1}(\underline{\mathbf{x}})|$ in total.
We examine if it is beneficial to select $\underline{\mathbf{x}} \in \mathcal{D}^\star$ that has most 1-Neighbours in the first iteration. In other words, in the first iteration we select $\underline{\mathbf{x}} \in \mathcal{D}^\star$ such that $|N_1(\underline{\mathbf{x}})| = \max\{|N_1(\underline{\mathbf{x}})| : \underline{\mathbf{x}} \in \mathcal{D}^\star\}$.
From the second iteration onwards, the key $\underline{\mathbf{x}} \in \mathcal{D}^\star$ of lowest lexicographical order from $S_{\max}$ is selected as in reference Algorithm 1.

A similar adaptation was made to the first iteration of reference Algorithm 2: The reference Algorithm 2 deletes the element $\underline{\mathbf{x}} \in \mathcal{D}^\star$ of lowest lexicographical order from $S_{\min}$ from $\mathcal{D}$ and appends it to $\mathcal{C}$ in each iteration. This selection of $\underline{\mathbf{x}}$ will be adapted in the first iteration. We compare sizes of the subsets $N_j(\underline{\mathbf{x}}) \subseteq N_{d-1}(\underline{\mathbf{x}})$, structured in equation (39). We examine if it has potential to select keys $\underline{\mathbf{x}} \in \mathcal{D}^\star$ that have small j-Neighbourhoods $N_j(\underline{\mathbf{x}}) \subseteq N_{d-1}(\underline{\mathbf{x}})$ for some $j \in \{1, ..., (d-1)\}$ instead of selecting $\underline{\mathbf{x}} \in \mathcal{D}^\star$ from S_{min}, that has a small $|N_{d-1}(\underline{\mathbf{x}})|$ in total.
We examine if it is beneficial to select $\underline{\mathbf{x}} \in \mathcal{D}^\star$ that has least 1-Neighbours in the first iteration. In other words, in the first iteration we select $\underline{\mathbf{x}} \in \mathcal{D}^\star$ such that $|N_1(\underline{\mathbf{x}})| = \min\{|N_1(\underline{\mathbf{x}})| : \underline{\mathbf{x}} \in \mathcal{D}^\star\}$.
From the second iteration onwards, the key $\underline{\mathbf{x}} \in \mathcal{D}^\star$ of lowest lexicographical order from $S_{\min}$ is selected as in reference Algorithm 2.

These adaptations did not provide larger DNA-d codes than the ones that are generated by reference Algorithms 1 and 2, of which the sizes are represented in the first two columns of Tables 14 and 15. So we conclude the considered approach that selects keys $\underline{\mathbf{x}} \in \mathcal{D}^\star$ with large / small j-Neighbourhoods for some $j \in \{1, ..., (d-1)\}$ is not promising for finding larger lower bounds for $B_1(n, \lfloor \frac{n}{2} \rfloor, d)$.

6.2.2 A special case

It is obtained from Table 15 that there is one of the examined cases where reference Algorithm 3 gives a higher lower bound for $B_1(n, \lfloor \frac{n}{2} \rfloor, d)$ than reference Algorithm 2: the case where $n = 6$ and $d = 4$. The case is marked red in Table 15.

We studied the iterations of the reference Algorithms 2 and 3 closely for this combination of constraints. Large differences in lexicographical order between adjacent DNA-words in DNA-d code $\mathcal{C}$ generated by reference Algorithm 2 might explain a disadvantage compared to reference Algorithm 3, as the latter creates a DNA-d code in lexicographical order. It turned out the biggest difference in lexicographical order between adjacent DNA-words in *newcode* $\mathcal{C}$, generated by reference Algorithm 2, was between the first and the second DNA-word of $\mathcal{C}$.
Also, the first words of the DNA-d codes generated by reference Algorithms 2 and 3 respectively, are identical. This implies the first DNA-word of $\mathcal{B}_1(n, \lfloor \frac{n}{2} \rfloor)$, that is always the first element of a DNA-d code generated by reference Algorithm 3, also has the least $d-1$-Neighbours in the case $n = 6$, $d = 4$.

We examined whether the combination of these phenomena or something similar appeared more often. For these cases, we focused on a consistent possible positive effect for the sizes of DNA-d codes generated by reference Algorithm 3. No such effect in case of appearance of the described phenomena was found. We concluded the large code size of the DNA-d code generated by reference Algorithm 3 in the case $n = 6$, $d = 4$ is due to other reasons. Now that we know from reference Algorithm 3 that $B_1(6, 3, 4) \geq 21$, as obtained in Table 15, we consider adaptations to reference Algorithm 2 to obtain a DNA-d code of this size or larger.

6.3 The choice from $S_{\min}$

From Subsection 6.2.1, we conclude selecting DNA-words $\underline{\mathbf{x}}$ with large / small 1-Neighbourhoods in the first iteration is not promising. From Subsection 6.2.2, we conclude looking at the difference in lexicographical order of adjacent DNA-words in the DNA-d code generated by reference Algorithm 2 is not defining for the performance of the algorithm. As obtained from the first three columns of Tables 14 and 15, reference Algorithm 2 mostly generates the largest DNA-d codes.
If we look into the structure and principles of this algorithm, the attention is drawn to the choice of $\underline{\mathbf{x}} \in \mathcal{D}^\star$ in the first step of the iteration. This is the element of lowest lexicographical order in $S_{\min}$, referring to Definition 6.5. We consider choosing the DNA-word $\underline{\mathbf{x}}$ from $S_{\min}$, if $|S_{\min}| > 1$, differently in this step of the iteration.

6.3.1 Algorithm 4 on lexicographical potential

Throughout this Subsection, we consider $S_{\min}$, as defined in Definition 6.5, with its elements represented in lexicographical order. In reference Algorithm 2, the first element of $S_{\min} \subseteq \mathcal{D}_2^\star$ is selected in each iteration.
We examine the potential of a certain lexicographic balance: we design an Algorithm in which we select the DNA-word halfway $S_{\min}$, defined as $\underline{\mathbf{s}}_h$:

Definition 6.6. Consider $S_{\min} = \{\underline{\mathbf{s}}_1, \underline{\mathbf{s}}_2, ...\underline{\mathbf{s}}_{|S|}\}$ where $|S| = |S_{\min}|$ as defined in Definition 6.5. We define $\underline{\mathbf{s}}_h$ as:

$$\underline{\mathbf{s}}_h \in S_{\min},\ h = \left\lfloor \frac{|S|}{2} \right\rfloor. \tag{40}$$

Mind that this DNA-word still has the least $d-1$ Neighbours, so we adhere to compliance with the principles of reference Algorithm 2. We obtain in Table 14 and Table 15, if we compare the column of Algorithm 4 with the previous ones, that larger DNA-d codes are generated, hence this approach is beneficial in many cases. We also considered taking the last element of $S_{\min}$, but no improved lower bound for $B_1(n, \lfloor \frac{n}{2} \rfloor, d)$ was found from sizes of the generated DNA-d codes, compared with the results in Tables 14 and 15.

Algorithm 4 Creating a DNA-d code by selecting DNA-words halfway $S_{\min}$. Returning a lower bound for $B_1(n, \lfloor \frac{n}{2} \rfloor, d)$.

Input: Constraints:

- maximum runlength $r = 1$;
- DNA-word length n;
- weight $w = \lfloor \frac{n}{2} \rfloor$;
- minimum Hamming distance d.

Output: A lower bound for $B_1(n, \lfloor \frac{n}{2} \rfloor, d)$.
Data:

- A lexicographically ordered dictionary *words_in_sphere* $\mathcal{D}_1 = \{\{\underline{\mathbf{x}}_i : N_{d-1}(\underline{\mathbf{x}}_i)\} : i \in \{1, ..., B_1(n, \lfloor \frac{n}{2} \rfloor)\}\}$;
- A lexicographically ordered dictionary *distances* $\mathcal{D}_2 = \{\{\underline{\mathbf{x}}_i : |N_{d-1}(\underline{\mathbf{x}}_i)|\} : i \in \{1, ..., B_1(n, \lfloor \frac{n}{2} \rfloor)\}\}$;
- An empty list *newcode* $\mathcal{C}$. ▷ See comment 1 below

```
18 Function Iteration(distances, words_in_sphere, newcode):
19     word = $\underline{\mathbf{s}}_h$;                                    ▷ See comment 2 below
       word is appended to newcode $\mathcal{C}$;
       word and the set $\{\underline{word} : \underline{word} \in N(word)\}$ are deleted from dictionaries $\mathcal{D}_1$ and $\mathcal{D}_2$;
       Both dictionaries $\mathcal{D}_1$ and $\mathcal{D}_2$ are updated;      ▷ See comment 3 below
       return distances $\mathcal{D}_2$, words_in_sphere $\mathcal{D}_1$, newcode $\mathcal{C}$.
20
21 Function Algorithm:
22     While $|\mathcal{D}_1| > 0$: Iteration;
23     return $|\mathcal{C}|$: a lower bound for $B_1(n, \lfloor \frac{n}{2} \rfloor, d)$.   ▷ See comment 4 below
```

Comments

1. This will eventually reperesent the generated code.
2. $\underline{\mathbf{s}}_h \in S_{\min} \subseteq \mathcal{D}_2^{\star}$ As defined in Definitions 6.5 and 6.6.
3. If *word* or $\underline{word}$ was a $d-1$-Neighbour of one of the keys that is still left in the dictionaries, the value of this key should be subtracted.
4. In the Python code in Appendix A this is referred to as the length of *newcode*.

6.3.2 Algorithm 5 on Neighbourhoods of Neighbourhoods

We discover the consequences of working with other properties than the lexicographical order of DNA-words in $S_{\min}$. We present an Algorithm that rests on the same principle as Algorithms 2 and 4, presented in Subsections 6.1.2 and 6.3.1 respectively, but selects from $S_{\min}$ in a more detailed way.
For each DNA-word $\underline{\mathbf{s}} \in S_{\min}$, we know it has the same, minimal number of $d-1$-Neighbours. The $d-1$-Neighbourhoods $N_{d-1}(\underline{\mathbf{s}})$ are equally large but not identical! We distinguish them as follows: we consider the sizes of the $d-1$-Neighbourhoods of the elements $\underline{\mathbf{y}} \in N_{d-1}(\underline{\mathbf{s}})$ and add them up:

Definition 6.7. We define the **size of the extended $d-1$-Neighbourhood of $\underline{\mathbf{s}} \in S_{\mathbf{min}}$** denoted as $|\mathbf{N}(\underline{\mathbf{s}})|$, as:

$$|N(\underline{\mathbf{s}})| = \sum_{i=1}^{|N_{d-1}(\underline{\mathbf{s}})|} |N_{d-1}(\underline{\mathbf{y}}_i)|, \ \underline{\mathbf{y}}_i \in N_{d-1}(\underline{\mathbf{s}}). \tag{41}$$

Note that we allow double counting when DNA-words appear in multiple $d-1$-Neighbourhoods $N_{d-1}(\underline{\mathbf{y}})$, where $\underline{\mathbf{y}} \in N_{d-1}(\underline{\mathbf{s}})$ and $\underline{\mathbf{s}} \in S_{\min}$.

The Algorithm selects the DNA-word $\underline{\mathbf{s}} \in S_{\min}$ with the largest extended $d-1$-Neighbourhood, defined as $\underline{\mathbf{s}}_e$:

$$\underline{\mathbf{s}}_e = \underline{\mathbf{s}}_e \in S_{\min} : |N(\underline{\mathbf{s}}_e)| = \max\{|N(\underline{\mathbf{s}})| : \ \underline{\mathbf{s}} \in S_{\min}\}. \tag{42}$$

Where $|N(\underline{\mathbf{s}}_e)|$ is defined according to Definition 6.7.

It is obtained from the fifth column of Tables 14 and 15 respectively, that almost all lower bounds for $B_1(n, \lfloor \frac{n}{2} \rfloor, d)$ returned by this Algorithm, are equal to or higher than the ones from Algorithms 1 up to and including 4. This holds for all examined cases but $n = 7$ in Table 14, where Algorithm 4 generates a larger DNA-3 code.

Algorithm 5 Creating a DNA-d code by selecting DNA-words $\underline{\mathbf{s}} \in S_{\min}$ with the largest extended $d-1$-Neighbourhood. Returning a lower bound for $B_1(n, \lfloor \frac{n}{2} \rfloor, d)$.

Input: Constraints:

- maximum runlength $r = 1$;
- DNA-word length n;
- weight $w = \lfloor \frac{n}{2} \rfloor$;
- minimum Hamming distance d.

Output: A lower bound for $B_1(n, \lfloor \frac{n}{2} \rfloor, d)$.
Data:

- A lexicographically ordered dictionary *words_in_sphere* $\mathcal{D}_1 = \{\{\underline{\mathbf{x}}_i : N_{d-1}(\underline{\mathbf{x}}_i)\} : i \in \{1, ..., B_1(n, \lfloor \frac{n}{2} \rfloor)\}\}$;
- A lexicographically ordered dictionary *distances* $\mathcal{D}_2 = \{\{\underline{\mathbf{x}}_i : |N_{d-1}(\underline{\mathbf{x}}_i)|\} : i \in \{1, ..., B_1(n, \lfloor \frac{n}{2} \rfloor)\}\}$;
- An empty list *newcode* $\mathcal{C}$. ▷ See comment 1 below

```
24 Function Iteration(distances, words_in_sphere, newcode):
25     $word = \underline{\mathbf{s}}_e$                                   ▷ See comment 2 below;
       $word$ is appended to $newcode$ $\mathcal{C}$;
       $word$ and the set $\{\underline{word} : \underline{word} \in N(word)\}$ are deleted from dictionaries $\mathcal{D}_1$ and $\mathcal{D}_2$;
       Both dictionaries $\mathcal{D}_1$ and $\mathcal{D}_2$ are updated;   ▷ See comment 3 below
       return distances $\mathcal{D}_2$, words_in_sphere $\mathcal{D}_1$, newcode $\mathcal{C}$.
26
27 Function Algorithm:
28     While $|\mathcal{D}_1| > 0$: Iteration;
29     return $|\mathcal{C}|$: a lower bound for $B_1(n, \lfloor \frac{n}{2} \rfloor, d)$.   ▷ See comment 4 below
```

Comments

1. This will eventually represent the generated code.
2. $\underline{\mathbf{s}}_e \in S_{\min}$ as defined in equation (42). The algorithm automatically selects the DNA-word $\underline{\mathbf{s}}_e$ of lowest lexicographical order in $S_{\min}$ satisfying $|N_{d-1}(\underline{\mathbf{s}}_e)| = \max\{(|N_{d-1}(\underline{\mathbf{s}})| : \underline{\mathbf{x}} \in S_{\min}\}$.
3. If *word* or $\underline{word}$ was a $d-1$-Neighbour of one of the keys that is still left in the dictionaries, the value of this key should be subtracted.
4. In the Python code in Appendix A this is referred to as the length of *newcode*.

6.3.3 Looking back Algorithm 6

We obtain the outcomes of Algorithms 4 and 5 in the fourth and fifth columns of Tables 14 and 15 respectively. In order to improve the obtained lower bounds for $B_1(n, \lfloor \frac{n}{2} \rfloor, d)$ we present a self-critical Algorithm that selects DNA-words from $S_{\min}$ based on the previous iteration. This is possible in every iteration except in the first iteration. For the first iteration, every possible $\underline{\mathbf{s}} \in S_{\min}$ is considered as the first DNA-word of $\mathcal{C}$ and the optimal outcomes are presented in the sixth column of Tables 14 and 15. From the second iteration onwards, we select the DNA-word from $S_{\min}$ that has the largest Hamming distance to reference DNA-word $\underline{\mathbf{c}}_{\text{ref}}$:

Definition 6.8. Consider *newcode* $\mathcal{C}$ after iteration i: $\mathcal{C} = \{\underline{\mathbf{c}}_1, \underline{\mathbf{c}}_2, ..., \underline{\mathbf{c}}_i\}$. We define the **reference DNA-word $\underline{\mathbf{c}}_{\mathbf{ref}}$** as $\underline{\mathbf{c}}_{\text{ref}} = \underline{\mathbf{c}}_i$.

Definition 6.9. In iteration $i+1$, we define the **choice $\underline{\mathbf{s}}_c$** as:

$$\underline{\mathbf{s}}_c = \underline{\mathbf{s}}_c \in S_{\min} : \; H(\underline{\mathbf{s}}_c, \underline{\mathbf{c}}_{\text{ref}}) = \max\{H(\underline{\mathbf{s}}, \underline{\mathbf{c}}_{\text{ref}}) : \underline{\mathbf{s}} \in S_{\min}\}. \tag{43}$$

From the sixth column of Tables 14 and 15 it can be observed that for all examined cases except for the case $n = 8$, $d = 3$, Algorithm 6 generates DNA-d codes of sizes that achieve or surpass the sizes of the DNA-d codes generated by Algorithms 1 up to and including Algorithm 5.

Algorithm 6 Creating a DNA-d code by selecting DNA-words $\underline{\mathbf{s}} \in S_{\min}$ in each iteration with maximum Hamming distance to the selected DNA-word in previous iteration. Returning a lower bound for $B_1(n, \lfloor \frac{n}{2} \rfloor, d)$.

Input: Constraints:

- maximum runlength $r = 1$;
- DNA-word length n;
- weight $w = \lfloor \frac{n}{2} \rfloor$;
- minimum Hamming distance d.

Output: A lower bound for $B_1(n, \lfloor \frac{n}{2} \rfloor, d)$.
Data:

- A lexicographically ordered dictionary *words_in_sphere* $\mathcal{D}_1 = \{\{\underline{\mathbf{x}}_i : N_{d-1}(\underline{\mathbf{x}}_i)\} : i \in \{1, ..., B_1(n, \lfloor \frac{n}{2} \rfloor)\}\}$;
- A lexicographically ordered dictionary *distances* $\mathcal{D}_2 = \{\{\underline{\mathbf{x}}_i : |N_{d-1}(\underline{\mathbf{x}}_i)|\} : i \in \{1, ..., B_1(n, \lfloor \frac{n}{2} \rfloor)\}\}$;
- An empty list *newcode* $\mathcal{C}$. ▷ See comment 1 below

```
30 Function Iteration(distances, words_in_sphere, newcode):
31     word = $\underline{\mathbf{s}}_c$                                              ▷ See comment 2 below;
       word is appended to newcode $\mathcal{C}$;
       word and the set $\{\underline{word} : \underline{word} \in N(word)\}$ are deleted from dictionaries $\mathcal{D}_1$ and $\mathcal{D}_2$;
       Both dictionaries $\mathcal{D}_1$ and $\mathcal{D}_2$ are updated;           ▷ See comment 3 below
       return distances $\mathcal{D}_2$, words_in_sphere $\mathcal{D}_1$, newcode $\mathcal{C}$.
32
33 Function Algorithm:
34     While $|\mathcal{D}_1| > 0$: Iteration;
35     return $|\mathcal{C}|$: a lower bound for $B_1(n, \lfloor \frac{n}{2} \rfloor, d)$.   ▷ See comment 4 below
```

Comments for Algorithm 6

1. This will eventually represent the generated code.
2. $\underline{\mathbf{s}}_c \in S_{\min}$ as defined in equation (43). The algorithm automatically selects the DNA-word $\underline{\mathbf{s}}_c$ of lowest lexicographical order in $S_{\min}$ satisfying $H(\underline{\mathbf{s}}_c, \underline{\mathbf{c}}_{\text{ref}}) = \max\{H(\underline{\mathbf{s}}, \underline{\mathbf{c}}_{\text{ref}}) : \underline{\mathbf{s}} \in S_{\min}\}$.
3. If *word* or $\underline{word}$ was a $d-1$-Neighbour of one of the keys that is still left in the dictionaries, the value of this key should be subtracted.
4. In the Python code in Appendix A this is referred to as the length of *newcode*.

6.4 Results

We present the outcomes of the Algorithms described in previous Subsections 6.1 up to 6.3.3. The highest lower bounds for $B_1(n, \lfloor \frac{n}{2} \rfloor, d)$ for the examined cases are underlined.

Table 14: Lower bounds for $B_1(n, \lfloor \frac{n}{2} \rfloor, 3)$ from Algorithms

	Algorithm	1	2	3	4	5	6, outcome and first iteration
n	$\|\mathcal{C}_3\|$						
3		2	3	2	3	3	3
4		11	12	7	9	12	12, $\underline{\mathbf{s}}_1$
5		17	18	18	18	18	18, $\underline{\mathbf{s}}_1$
6		44	53	45	52	53	53, $\underline{\mathbf{s}}_3$
7		110	119	101	124	123	$\underline{127}$, $\underline{\mathbf{s}}_4$
8		289	326	286	330	$\underline{347}$	341, $\underline{\mathbf{s}}_1$
9		662	762	687	760	767	$\underline{783}$, $\underline{\mathbf{s}}_4$

Table 15: Lower bounds for $B_1(n, \lfloor \frac{n}{2} \rfloor, 4)$ from Algorithms

	Algorithm	1	2	3	4	5	6, outcome and first iteration
n	$\|\mathcal{C}_4\|$						
4		4	4	4	4	4	4
5		7	7	7	7	7	7, $\underline{\mathbf{s}}_1$
6		16	20	21	19	22	22, $\underline{\mathbf{s}}_1$
7		36	44	37	40	42	$\underline{44}$, $\underline{\mathbf{s}}_{11}$
8		86	127	97	128	128	128, $\underline{\mathbf{s}}_2$
9		199	227	216	231	235	235, $\underline{\mathbf{s}}_2$

7 Conclusions and Recommendations

As stated in Subsection 1.2, this research is performed to answer the question:

"Can we create DNA-codes and therefore determine upper bounds and lower bounds for the size of DNA-codes satisfying $d > 2$?"

Recall from Chapter 4 that $B_1(4,2,3) = 12$. We conclude from the fifth and sixth columns from Table 14 that Algorithm 5 and Algorithm 6 create the largest possible DNA-3 codes.

Recall from Chapter 5 that

$$\forall r, n \geq 1,\ B_r(n,w,n) = \begin{cases} 4 & w = \frac{n}{2} \text{ for } n \text{ even},\ n \neq \frac{n}{2}, \\ 3 & w \in \{\lceil \frac{n}{3} \rceil, ..., \lfloor \frac{2n}{3} \rfloor\}, w \neq \frac{n}{2}, \\ 2 & 0 \leq w < \lceil \frac{n}{3} \rceil \text{ or } \lfloor \frac{2n}{3} \rfloor < w \leq n. \end{cases} \tag{44}$$

From Table 14 it is concluded that only Algorithm 1 and Algorithm 3 do not generate the largest possible DNA-3 codes. From Table 15 it is concluded that Algorithms 1 up to and including Algorithm 6 create the largest possible DNA-4 codes.

We analyze the other outcomes as well and present suggestions for future research on analytical results as well as improvements of the Algorithms in the search for DNA-d codes of the largest possible size.

7.1 Sizes of the largest DNA-d codes generated by the Algorithms

We present the highest lower bounds for $B_1(n, \lfloor \frac{n}{2} \rfloor, d)$ obtained from the six algorithms presented in Chapters 6.1 up to 6.3.3.

Table 16: Highest obtained lower bounds for $B_1(n, \lfloor \frac{n}{2} \rfloor, d)$ from presented Algorithms

n	$\lvert\mathcal{C}_3\rvert$	$\lvert\mathcal{C}_4\rvert$
3	3	
4	12	4
5	18	7
6	53	22
7	127	44
8	347	128
9	783	235

.

7.2 Continuation in analytical research

As described in Chapter 2, DNA-d codes that satisfy a high imposed minimum Hamming distance d are less error prone when applying error detection or -correction decoding mechanisms. The higher the imposed d, the more potential sequencing errors can be guaranteed to be detected / corrected. To keep DNA-d codes information dense, we search for the largest possible DNA-d codes satisfying this high imposed d.
Analytical proofs like in Chapter 5 contribute in the determination of DNA-d code sizes significantly. The analytical results provide upper bounds as well as lower bounds, hence indisputable code sizes. Continuing analytical research up to high lengths n of DNA-words and high imposed minimum Hamming distances d of DNA-d codes improve the level of analysis we can apply to the outcome of algorithms. Also, the research would substantiate the reasoning behind the principles these algorithms adhere to. A suggestion would be building on Chapter 5 and consider the case $d = n - 1$.

7.3 Suggested improvements of the algorithms

We suggest improvements of the Algorithms presented in Subsections 6.1 up to and including 6.3.3.

7.3.1 Profound choices in algorithms

In each iteration of the presented algorithms various choices are made. These choices can be made more profound: an example of this is specifying the choice from S_{min}, like we did in Subsections 6.3.1 up to and including 6.3.3.
Consider the choices of $\underline{\mathbf{s}}_e$ and $\underline{\mathbf{s}}_c$ in Algorithms 5 and 6: the algorithms automatically select these DNA-words of lowest lexicographical order in S_{min} satisfying the desired properties. Algorithms that adhere to various ways of choosing from S_{min} can be discussed.

7.3.2 A special case for Algorithm 5

The underlined green outcomes in Tables 14 and 16 show an exceptionally high outcome of the size of the DNA-3 codes $\mathcal{C}_3 \subseteq \mathcal{B}_1(8,4)$ generated by Algorithm 5. It is recommended to look into this case in a similar way as we did in Subsection 6.2.2. Note that it will be hard to manually compare the generated DNA-3 codes as a whole, as they host over 300 DNA-words.
At first view, the outcomes of Algorithm 6 based on different choices from S_{min} in the first iteration are remarkably structured compared to the other examined cases:

Table 17: The choice from S_{min} in the first iteration and the corresponding outcome of Algorithm 6

$\underline{\mathbf{s}}_c$	$\lvert\mathcal{C}_3\rvert$
$\underline{\mathbf{s}}_1$	341
$\underline{\mathbf{s}}_2$	341
$\underline{\mathbf{s}}_3$	341
$\underline{\mathbf{s}}_4$	341
$\underline{\mathbf{s}}_5$	330
$\underline{\mathbf{s}}_6$	330
$\underline{\mathbf{s}}_7$	330
$\underline{\mathbf{s}}_8$	330

.

If the algorithm is applied for higher n for example and this structure of S_{min} in the first iteration appears more often, it can be checked if Algorithm 5 is beneficial over Algorithm 6 in these cases as well. Consistencies and new thoughts on algorithm designs can arise from analysis of similar cases in the search for the largest possible DNA-d codes.

References

[1] X. Li B. Cao, S. Zhao and B. Wang. K-means Multi-Verse Optimizer (KMVO) Algorithm to Construct DNA Storage Codes. *IEEE Access, vol. 8, pp. 29547-29556*, 2020.

[2] Eva Garcia-Ruiz Jian Ma Huimin Zhao S. M. Hossein Tabatabaei Yazdi, Han Mao Kiah and Olgica Milenkovic. DNA Based Storage: Trends and Methods. *IEEE Transactions on Molecular, Biological and Multi-Scale Communications*, 1(3), 2015.

[3] Jos H. Weber. On Single-Error-Detecting Codes for DNA-Based Data Storage. *IEEE Communication Letters*, 25(1), 2021.

[4] C.J. (Lot) van Leeuwen. Constrained Codes for DNA-Based Storage Systems, 2020.

[5] H. Vermeer. Constrained Single-Error-Detecting codes for DNA-based Storage Systems, 2021.

[6] Vaneet Aggarwal Dixita Limbachiya, Manish K. Gupta. Family of Constrained Codes for Archival DNA Data Storage. *IEEE Communications Letters*, 22(10), 2018.

[7] Jeff Desjardins. How much Data is generated each day? `https://www.weforum.org/agenda/2019/04/how-much-data-is-generated-each-day-cf4bddf29f/`, 2019.

[8] Zijn er meer sterren in het heelal dan zandkorrels op aarde? `https://www.astronomie.nl/veelgestelde-vragen/zijn-er-meer-sterren-in-het-heelal-dan-zandkorrels-op-aarde-20?category=Vragen+over+sterren+en+sterrenstelsels`, date of consult: december 2021.

[9] E. M. Rubin an S. Kosuri G. M. Church. Next-Generation Digital Information Storage in DNA. *Science (New York, N. Y.)*, 337(6102, p. 1628), 2021.

[10] Y. Erlich and D. Zielinski. DNA Fountain enables a robust and efficient storage architecture. *Science*, 355, 2016.

[11] M. Puddu D. Paunescu R. N. Grass, R. Heckel and W. J. Stark. Robust Chemical Preservation of Digital Information on DNA in Silica with Error-Correcting Codes. *Angewandte Chemie*, 54(8, p.2552-2555), 2015.

[12] Bas van Rijzewijk. Interview with Kees Schouhamer Immink. *De Zaak*, september 2020. `https://www.dezaak.nl/magazine/pioniers/het-draaide-meer-om-de-wedstrijd-die-ik-meemaakte/`, *translated.*

[13] Kui Cai Kees A. Schouhamer Immink. Efficient Balanced and Maximum Homopolymer-Run Restricted Block Codes for DNA-Based Data Storage. *IEEE Communications Letters*, 23(10), 2019.

[14] The Editors of Encyclopaedia Britannica. DNA. 2020. Accessed December 2021.

[15] K. Maeda E. Yashima and Y. Furusho. Single- and Double-Stranded Helical Polymers: Synthesis, Structures, and Functions. *Accounts of chemical research*, 41(9), 2008.

[16] T. Chauhan. DNA Sequencing: History, Steps, Methods, Applications and Limitations. 2019. Accessed December 2021.

[17] J.H. Weber. Lecture Notes: Error-Correcting Codes, April 2019.

[18] Oliver D. King. Bounds for DNA Codes with Constant CG-content. *The electronic journal of Combinatorics*, 10, 2003.

[19] A. E. CONDON A. MARATHE and R. M. CORN. On Combinatorial DNA Word Design. *Journal of Computational Biology*, 8(3), 2001.

[20] K. A. Schouhamer Immink. Runlength-limited sequences. *Proceedings of the IEEE*, 78(11, pp. 1745–1759), Nov 1990.

[21] M. Costello et al. M.G. Ross, C. Russ. Characterizing and Measuring Bias in Sequence Data. *Genome Biology*, 14(R51), 2013.

[22] Márquez-Corbella. Code-based Cryptography: Error-Correcting Codes and Cryptography. *Institut national de recherche en informatique et en automatique (INRIA)*, 2016.

A Python codes

A.1 Reference algorithms

```python
import itertools as it

length = int(input('wordlength='))
symbols = '0123'
weight = int(length/2)
runlength = 1

def list_words(length, symbols):
    return [''.join(x) for x in it.product(symbols, repeat=length)]


allwords = list_words(length, symbols)

weighted_words = []
for word in allwords:
    w = word.count('2') + word.count('3')
    if w == weight:
        weighted_words.append(word)

runlength_words = []

for word in weighted_words:
    repeat_counter = 1
    illegal_word = False
    for i in range(1, length):
        if word[i] == word[i - 1]:
            repeat_counter += 1
        else:
            repeat_counter = 1

        if repeat_counter > runlength:
            illegal_word = True
            break

    if not illegal_word:
        runlength_words.append(word)

length_words = []

for word in runlength_words:
    if len(word) == length:
        length_words.append(word)


finalwords=length_words

print('length final words is ', len(finalwords))

def get_distance(word1,word2):
    #determine the distance between two words
    return len([i for i in range(len(word1)) if
                int(word1[i])-int(word2[i]) != 0])


def get_words_in_sphere(code,codeword, d):
    #get all the (d-1)-neighbours of a codeword
    return [w for w in code if 0<get_distance(codeword,w)<d]

d=int(input("distance="))
code=finalwords

def dictionaryNBH(code, d):
    words_in_sphere={codeword: get_words_in_sphere(code, codeword, d)
                     for codeword in code}
    return words_in_sphere
```

```

words_in_sphere=dictionaryNBH(code, d)

def dictionarydist(words_in_sphere):
    distances_in_sphere={key:len(value) for key, value in words_in_sphere.items()}
    return distances_in_sphere

distances=dictionarydist(words_in_sphere)

#start iteration
#ALG 1
def iteration1(distances, words_in_sphere):
    #words with most d-1 NBs
    maximum = max(distances, key=distances.get)
    del distances[maximum]
    for value in words_in_sphere[maximum]:
        distances[value]-=1
        words_in_sphere[value].remove(maximum)
    del words_in_sphere[maximum]

def alg1(distances, words_in_sphere):
    while max(distances.values())>0:
        iteration1(distances, words_in_sphere)
    return len(words_in_sphere)

alg1=alg1(distances, words_in_sphere)
print('Lowerbound of size with algorithm 1 and distance ', d, ' is', alg1)

words_in_sphere=dictionaryNBH(code, d)
distances=dictionarydist(words_in_sphere)

#ALG 2
newcode=[]
def iteration2(distances, words_in_sphere, newcode):
    minimum = min(distances, key=distances.get)
    newcode.append(minimum)
    del distances[minimum]
    for value in words_in_sphere[minimum]:
        del distances[value]
        for val in words_in_sphere[value]:
            if val in distances:
                words_in_sphere[val].remove(value)
                distances[val]-=1
        del words_in_sphere[value]
    del words_in_sphere[minimum]
    return newcode

def alg2(distances, words_in_sphere, newcode):
    while words_in_sphere:
        iteration2(distances, words_in_sphere, newcode)
    return len(newcode)

alg2=alg2(distances, words_in_sphere, newcode)
print("\nLowerbound of size with algorithm 2 and distance", d, 'is', alg2)

#ALG 3
DNAcode=finalwords
alg3code=[]
def iteration3(alg3code, DNAcode, d):
    word=DNAcode[0]
    if all(get_distance(word, codeword)>=d for codeword in alg3code):
        alg3code.append(word)
    DNAcode.remove(word)
    return alg3code, DNAcode

def alg3(alg3code, DNAcode, d):
    while DNAcode:
        iteration3(alg3code, DNAcode, d)
    return len(alg3code)
```

```
alg3=alg3(alg3code, DNAcode, d)
print('\nLowerbound of size with algorithm 3 and distance', d, 'is', alg3)
```

A.2 Algorithm 4

```
import itertools as it

length = int(input('wordlength='))
symbols = '0123'
weight = int(length/2)
runlength = 1

def list_words(length, symbols):
    return [''.join(x) for x in it.product(symbols, repeat=length)]


allwords = list_words(length, symbols)

weighted_words = []
for word in allwords:
    w = word.count('2') + word.count('3')
    if w == weight:
        weighted_words.append(word)

runlength_words = []

for word in weighted_words:
    repeat_counter = 1
    illegal_word = False
    for i in range(1, length):
        if word[i] == word[i - 1]:
            repeat_counter += 1
        else:
            repeat_counter = 1

        if repeat_counter > runlength:
            illegal_word = True
            break

    if not illegal_word:
        runlength_words.append(word)

length_words = []

for word in runlength_words:
    if len(word) == length:
        length_words.append(word)


finalwords=length_words

print('length final words is ', len(finalwords))

def get_distance(word1,word2):
    #determine the distance between two words
    return len([i for i in range(len(word1)) if
                int(word1[i])-int(word2[i]) != 0])


def get_words_in_sphere(code,codeword, d):
    #get all the (d-1)-neighbours of a codeword
    return [w for w in code if 0<get_distance(codeword,w)<d]

d=int(input("distance="))
code=finalwords

def dictionaryNBH(code, d):
    words_in_sphere={codeword: get_words_in_sphere(code, codeword, d)
                     for codeword in code}
    return words_in_sphere

words_in_sphere=dictionaryNBH(code, d)

def dictionarydist(words_in_sphere):
    distances_in_sphere={key:len(value) for key, value in words_in_sphere.items()}
    return distances_in_sphere

distances=dictionarydist(words_in_sphere) #dictionary; woorden: aantal d-1 buren

#ALG

newcode=[]
def iteration2(distances, words_in_sphere, newcode):
    minlst = []
    minimum = min(distances, key=distances.get)
    for elt in distances:
        if distances[elt]==distances[minimum]:
            minlst.append(elt)
    minimum=minlst[int(len(minlst)/2)]
    newcode.append(minimum)
    del distances[minimum]
    for value in words_in_sphere[minimum]:
        del distances[value]
        for val in words_in_sphere[value]:
            if val in distances:
                words_in_sphere[val].remove(value)
                distances[val]-=1
        del words_in_sphere[value]
    del words_in_sphere[minimum]
    return newcode

def alg2(distances, words_in_sphere, newcode):
    while words_in_sphere:
        iteration2(distances, words_in_sphere, newcode)
    return len(newcode)

alg2=alg2(distances, words_in_sphere, newcode)
print("\nLowerbound of size with the algorithm and distance", d, 'is', alg2)
```

A.3 Algorithm 5

```
import itertools as it

length = int(input('wordlength='))
symbols = '0123'
weight = int(length/2)
runlength = 1

def list_words(length, symbols):
    return [''.join(x) for x in it.product(symbols, repeat=length)]


allwords = list_words(length, symbols)

weighted_words = []
for word in allwords:
    w = word.count('2') + word.count('3')
    if w == weight:
        weighted_words.append(word)

runlength_words = []

for word in weighted_words:
    repeat_counter = 1
    illegal_word = False
    for i in range(1, length):
        if word[i] == word[i - 1]:
            repeat_counter += 1
        else:
            repeat_counter = 1

        if repeat_counter > runlength:
            illegal_word = True
            break

    if not illegal_word:
        runlength_words.append(word)

length_words = []

for word in runlength_words:
    if len(word) == length:
        length_words.append(word)


finalwords=length_words

print('length final words is ', len(finalwords))

def get_distance(word1,word2):
    #determine the distance between two words
    return len([i for i in range(len(word1)) if
                int(word1[i])-int(word2[i]) != 0])


def get_words_in_sphere(code,codeword, d):
    #get all the (d-1)-neighbours of a codeword
    return [w for w in code if 0<get_distance(codeword,w)<d]

d=int(input("distance="))
code=finalwords

def dictionaryNBH(code, d):
    words_in_sphere={codeword: get_words_in_sphere(code, codeword, d)
                     for codeword in code}
    return words_in_sphere

words_in_sphere=dictionaryNBH(code, d) #dictionary; woorden: d-1 buren

def dictionarydist(words_in_sphere):
    distances_in_sphere={key:len(value) for key, value in words_in_sphere.items()}
    return distances_in_sphere

distances=dictionarydist(words_in_sphere) #dictionary; woorden: aantal d-1 buren

##print("distances = ", distances)

#ALG
newcode=[]
def iteration2(distances, words_in_sphere, newcode):
    minlst = []
    minimum = min(distances, key=distances.get)
    for elt in distances:
        if distances[elt]==distances[minimum]:
            minlst.append(elt)
    Nbs=[]
    for w in minlst:
        dlst=[]
        for w2 in words_in_sphere[w]:
            dlst.append(distances[w2])
        Nbs.append(dlst)
    num=max(sum(x) for x in Nbs)
    for lst in Nbs:
        if sum(lst)==num:
            index=Nbs.index(lst)
    minimum = minlst[index]
    newcode.append(minimum)
    del distances[minimum]
    for value in words_in_sphere[minimum]:
        del distances[value]
        for val in words_in_sphere[value]:
            if val in distances:
                words_in_sphere[val].remove(value)
                distances[val]-=1
        del words_in_sphere[value]
    del words_in_sphere[minimum]
    return newcode


def alg2(distances, words_in_sphere, newcode):
    while words_in_sphere:
        iteration2(distances, words_in_sphere, newcode)
    return len(newcode)

alg2=alg2(distances, words_in_sphere, newcode)
print("\nLowerbound of size with the algorithm and distance", d, 'is', alg2)
```

A.4 Algorithm 6

```
import itertools as it

length = int(input('wordlength='))
symbols = '0123'
weight = int(length/2)
runlength = 1

def list_words(length, symbols):
    return [''.join(x) for x in it.product(symbols, repeat=length)]


allwords = list_words(length, symbols)

weighted_words = []
for word in allwords:
    w = word.count('2') + word.count('3')
    if w == weight:
        weighted_words.append(word)

runlength_words = []

for word in weighted_words:
    repeat_counter = 1
    illegal_word = False
    for i in range(1, length):
        if word[i] == word[i - 1]:
            repeat_counter += 1
        else:
            repeat_counter = 1

        if repeat_counter > runlength:
            illegal_word = True
            break

    if not illegal_word:
        runlength_words.append(word)

length_words = []

for word in runlength_words:
    if len(word) == length:
        length_words.append(word)


finalwords=length_words

print('length final words is ', len(finalwords))
```

```
def get_distance(word1,word2):
    #determine the distance between two words
    return len([i for i in range(len(word1)) if
                int(word1[i])-int(word2[i]) != 0])


def get_words_in_sphere(code,codeword, d):
    #get all the (d-1)-neighbours of a codeword
    return [w for w in code if 0<get_distance(codeword,w)<d]

d=int(input("distance="))
code=finalwords

def dictionaryNBH(code, d):
    words_in_sphere={codeword: get_words_in_sphere(code, codeword, d)
                     for codeword in code}
    return words_in_sphere

words_in_sphere=dictionaryNBH(code, d)

def dictionarydist(words_in_sphere):
    distances_in_sphere={key:len(value) for key, value in words_in_sphere.items()}
    return distances_in_sphere

distances=dictionarydist(words_in_sphere) #dictionary; woorden: aantal d-1 buren

#ALG
newcode=[]
def iteration2(distances, words_in_sphere, newcode):
    minlst = []
    minimum = min(distances, key=distances.get)
    for elt in distances:
        if distances[elt]==distances[minimum]:
            minlst.append(elt)
    if newcode == []:
        minimum = minlst[6] #the index alters for different individual cases
    else:
        dlst=[]
        for option in minlst:
            d = get_distance(newcode[-1], option)
            dlst.append(d)
        maxd = max(dlst)
        index=dlst.index(maxd)
        minimum=minlst[index]
    newcode.append(minimum)
    del distances[minimum]
    for value in words_in_sphere[minimum]:
        del distances[value]
        for val in words_in_sphere[value]:
            if val in distances:
                words_in_sphere[val].remove(value)
                distances[val]-=1
        del words_in_sphere[value]
    del words_in_sphere[minimum]
    return newcode

def alg2(distances, words_in_sphere, newcode):
    while words_in_sphere:
        iteration2(distances, words_in_sphere, newcode)
    return len(newcode)

alg2=alg2(distances, words_in_sphere, newcode)
print("\nLowerbound of size with algorithm and distance", d, 'is', alg2)
```