Codes with relaxed weight constraints for DNA-based storage systems

by

J.M. Capel

to obtain the degree of Bachelor of Science
at the Delft University of Technology,

Student number:	4905180	
Project duration:	April, 2022 – August 24, 2022	
Thesis committee:	Dr. ir. J. H. Weber,	TU Delft, supervisor
	Dr. J. A. M. de Groot,	TU Delft, supervisor
	Drs. E. M. van Elderen,	TU Delft

An electronic version of this thesis is available at `http://repository.tudelft.nl/`.

TU Delft

Preface

This thesis is written as final part of the Bachelor Applied Mathematics at the Delft University of Technology. In the second year of this Bachelor I choose to take the elective course “Applied Algebra; Codes and Cryptography Systems” lectured by Dr. J.A.M. de Groot. This introduced me to coding theory, which made me realise I was very interested in this subject. Therefore I was immediately enthusiastic that Dr. ir. J.H. Weber gave me the chance to do my thesis in this field of study. I really like that this is a topic where it becomes very clear how mathematics can be used to find solutions for our future society.

I would like to thank my supervisors Dr. ir. Jos Weber and Dr. Joost de Groot for their guidance during this project. They gave me a lot of feedback and supported me during the whole process of writing this thesis. In addition, I would also like to thank Drs. E. M. van Elderen for being part of my Bachelor committee.

Judith Capel
Delft, July 2022

Abstract

Nowadays, we produce an enormous amount of data and all this data needs to be stored somewhere. In order to store all this data in an efficient and environmentally friendly way, solutions need to be found. A storage technique that has great potential to be a solution is synthetic DNA based storage. A DNA-strand is made up of a sequence of four nucleotides, Adenine (A), Thymine (T), Guanine (G) and Cytosine (C). To save our data using DNA, our binary data is encoded to quaternary data, using this quaternary data DNA strands are made. During this process, of saving our data in DNA, errors can occur, therefore two constraints are considered to minimise the number of errors in our codes. The GC-weight, which is the number of G and C nucleotides in a DNA word and the maximal run-length, which is the maximum number of identical nucleotides in a row. In addition, the Hamming distance between words is also considered to be able to detect and even correct errors that still occur.

This research tries to find lower and upper bounds for the maximum size of DNA codes. We continue the work of Van Leeuwen [1], Vermeer [2] and Laseur [3], who considered codes with fixed GC-weight. In this thesis we will relax this GC-weight constraint and we will look at codes where we have a set of GC-weights and all the DNA words in the code need to satisfy one of these weights. Furthermore, the codes we consider have a run-length of $r = 1$. We will determine the maximum size of codes where the minimum Hamming distance d is equal to the length n of the DNA words. We determine lower and upper bounds for the case where $d = n - 1$ and for specific sets of GC-weights we will determine the maximum size. Finally we will present nine algorithms, these algorithms create DNA-codes. The size of these codes give us lower bounds for the maximum size of DNA codes.

Contents

1 Introduction

1.1. Motivation

Nowadays, we produce an enormous amount of data, and all this data need to be stored somewhere. Right now, most of this data is stored on magnetic and optical media. Although there have been improvements in the storage capabilities of these media, significant advances in storage density and durability need to be sought, in order to preserve all our data [4]. Synthetic DNA based storage is costly, but has a great potential to be a solution to this problem.

Using magnetic and optical media as storage system has a density of about 100 GB$/mm^3$ [4]. Comparing this to DNA, DNA storage has a raw limit of 1 exabyte$/mm^3 = 10^9$ GB$/mm^3$[4]. So storing data in DNA would take up a lot less physical space. Furthermore the durability of DNA is also really good, we can still recover DNA from species that have been extinct for more than 10.000 years [5]. This while the durability of the current techniques is much worse. Rotating disks have a lifespan of 3-5 years and tape 10-30 years [4].

Although writing and reading DNA sequences is costly, DNA storage has great potential to be a technique for archival data storage [6]. In this thesis we will focus on the research on constraints in order to find quaternary code designs that can be used to save data using this promising technique.

1.2. Thesis statement

It is clear that, due to the durability of DNA and the fact that DNA has a dense storing capacity, DNA would be a very good potential storing technique. In order to eventually store data in DNA, research needs to be done into DNA codes. In [6] two constraints for DNA codes are described, DNA-words should satisfy these constraints to avoid errors. These constraints concern the fixed number of G and C nucleotides, the GC-weight, and the maximum number of identical nucleotides in a row, the run-length. To make the DNA codes less error prone, the words should contain no adjacent repeated nucleotides and the number of G and C nucleotides should be around half of the length of the DNA words. In addition to these two constraints, the minimum Hamming distance of a DNA code is also important. This allows errors to be detected and even corrected. As said to make the DNA words less prone the number of G and C nucleotides should be around half of the length of the DNA words. However, how does it effect the maximum size of codes if we don't create and analyze codes where all the words have the same fixed GC-weight, but where the words have to meet one of the GC-weights from a set with multiple different GC-weights. Building on the studies of [1], [2] and [3], in this thesis we address the question:

"Which upper and lower bounds can we determine for the maximum size of DNA-codes where the weight constraint is relaxed?"

We determine the maximum size of a code for the case where we have $d = n$ for all possible sets with GC-weights. We determine lower and upper bounds for the case where $d = n - 1$ and for specific sets of GC-weights we define the maximum size. Furthermore we will present algorithms, which create DNA-codes. The size of these codes give us lower bounds for the maximum size of DNA-codes.

1.3. Organisation of the Thesis:

In this section we will present how the remainder of this thesis is organised.

Chapter 2: Prerequisites: The first section of this chapter will give some basic knowledge about DNA. Thereafter it presents some basic concepts of DNA sets, these concepts will be used throughout this thesis.

Chapter 3: The maximum size of DNA-n codes: This chapter provides us with the maximum size of DNA codes where $d = n$. This chapter determines the maximum size of codes for all possible word-lengths n, all run-lengths r and all possible sets with GC-weights.

Chapter 4: The maximum size of DNA codes with $d = n - 1$: This chapter provides us with upper bounds for DNA codes where the words can't have two identical symbols following each other and where the symbols of the different words must be different at every position, except for one position. For all $n \geq 3$, except for $n = 5$, we determine the largest possible code size for a specific set $\mathcal{W}$. For the case where $n = 5$ also a lower bound is found and a range is given in which the largest possible size must be.

Chapter 5: Algorithms to compute DNA codes: This chapter gives algorithms from existing research [1] [3]. In addition, new algorithms are presented. The last section of this chapter evaluates the algorithms.

Chapter 6: Conclusions and recommendations: This final chapter will conclude the results found in the preceding chapters. Furthermore, it gives recommendations for further research in DNA based storage.

2

Prerequisites

2.1. What is DNA and how do we store data in DNA?

DeoxyriboNucleic Acid, abbreviated as DNA, is a macro molecule, which contains the biological information of an organism. DNA consists of two strands. These strands consist of four basic blocks, called nucleotides [4]. The four nucleotides are Adenine, Thymine, Guanine, and Cytosine. In this thesis, we will use the abbreviations A, T, G and C respectively. The two strands can bind to each other and form a double helix. This will happen when they are complementary. This means that the A in one string will bond with T in the other string, and likewise, C in one string will bond with G in the other string. This is called complementary base pairing [7].

When a DNA based storage system is used, there are a couple of stages which the data has to go through [6].
First the binary data needs to be encoded to quaternary data. Quaternary data is used because DNA is made up of the four nucleotides.
Then the data needs to be written on DNA, DNA strands are created using a DNA synthesizer. These DNA strands are stored until use.
Lastly when we want to use the data, the DNA strands need to be read. We read the data using DNA sequencing to get the quaternary data.
However the techniques used can lead to errors in the sequence of nucleotides. This means that the stored sequence is not the same as the sequence after the DNA strand is read. To make the sequences less prone, we consider constraints for the sequences. In the next section we will introduce these.

2.2. Basic DNA sets

In this section we will present some basic concepts of DNA sets, using sources [8] and [9] . Suppose we want to store some binary data using DNA, as mentioned in Section 2.1 we need to encode this over the quaternary alphabet $\{0, 1, 2, 3\}$. For convenience we will use the following bijection between the nucleotides and the quaternary alphabet:

$$A \leftrightarrow 0, T \leftrightarrow 1, G \leftrightarrow 2, C \leftrightarrow 3. \tag{2.1}$$

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

$$\mathbf{x} = [x_1, x_2, \dots, x_n] \tag{2.2}$$

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

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

$$\mathcal{B}(n) = \{\mathbf{x} : \mathbf{x} \text{ is a word of length } n\} \tag{2.3}$$

with cardinality

$$B(n) = |\mathcal{B}(n)| = 4^n. \tag{2.4}$$

Example 2.1. We will present an example with the set of all words of length 2.

$\mathcal{B}(n) = \{[0,0],[0,1],[0,2],[0,3],[1,0],[1,1],[1,2],[1,3],[2,0],[2,1],[2,2],[2,3],[3,0],[3,1],[3,2],[3,3]\}.$

We see that the cardinality of this set is indeed $4^2 = 16$.

$$B(n) = 16.$$

Definition 2.3. We define a **DNA-code**, denoted by $\mathcal{C}$, as a subset of $\mathcal{B}_n$. The words in this subset we call DNA-words. The size of a code is the cardinality of the set.

As mentioned in Section 2.1, when storing data in DNA errors can occur in the sequence of nucleotides. Constraints are considered to make the sequences less prone. An important source of errors in the DNA strands is the number of G and C nucleotides in the strand [6]. We say that the GC-weight of a DNA-word $\mathbf{x}$ is the number of G and C nucleotides in the word, denoted by $w(\mathbf{x})$. It is desirable to have a DNA-code where all the DNA-words have roughly the same GC-weight, because then the melting temperatures are similar [10].

Definition 2.4. We define the **GC-weight** $w(\mathbf{x})$ of a word $\mathbf{x}$ as the number of symbols in $\mathbf{x}$ that are equal to 2 or 3. We say

$$w_i = \begin{cases} 0 & \text{if } x_i = 0,1 \\ 1 & \text{if } x_i = 2,3 \end{cases} \tag{2.5}$$

and the GC-weight is defined as

$$w(\underline{\mathbf{x}}) = \sum_{i=1}^{n} w_i. \tag{2.6}$$

Example 2.2. Let us consider two words $\mathbf{x_1}$ and $\mathbf{x_2}$ where

$$\mathbf{x_1} = [0,1,0,2,3,2,1,0,1] \quad \text{and} \quad \mathbf{x_2} = [1,2,2,0,3]$$

then $\mathbf{x_1}$ is a word of length 9 and $\mathbf{x_2}$ is a word of length 5 and both have a weight $w(\mathbf{x_j}) = 3$ for $j \in \{1,2\}$.

Next to the GC-weight, we are also interested in the number of consecutive identical symbols in a word $\mathbf{x}$. A DNA-code satisfies a certain maximum run-length r, if for each code-word the maximum run-length is less than or equal to this r.

Definition 2.5. We define the **run-length** $r(\mathbf{x})$ of a word as the maximum number of subsequent identical symbols in $\mathbf{x}$:

$$r(\mathbf{x}) = \max\{r \,:\, \exists i \text{ such that } x_i = x_{i+1} = \cdots = x_{i+r-1}\}. \tag{2.7}$$

Example 2.3. Consider the word $\mathbf{x} = [0,0,1,2,2,2,3,1,2]$ we see that this word has run-length $r(\mathbf{x}) = 3$.

We are interested in DNA-codes where this constraint is small, because long runs of the same nucleotides causes errors. This is because during the DNA synthesis and sequencing, every DNA base is read as a signal. When we have a long run of the same nucleotides, the machine may read the long run of the same nucleotide as only one signal, and therefore the bases are lost during the sequencing. For example if we have the DNA-word $ATTTTGAC$. The machine may read the long run of T's as a smaller run. Thus, long runs of the same base, increases the loss rate of information and reduces the read coverage [6].

Now that we introduced the constraints, we can select words from the set $\mathcal{B}(n)$ that match a certain maximum run-length r and GC-weight w, these words together will form a subset of $\mathcal{B}_n$, which we call a DNA-code. In this thesis we will consider DNA-codes where it is not necessary that all the words in a code have the same GC-weight. We will consider codes where we allow w to be taken from a set $\mathcal{W}$. This set contains the GC-weights for which the words in the DNA-code we consider have to satisfy one of them.

Definition 2.6. We define the set $\mathcal{B}_r(n, \mathcal{W})$ as the set of all words $\mathbf{x}$ of length n that satisfy maximum run-length r and have weight $w \in \mathcal{W}$, i.e.,

$$\mathcal{B}_r(n, \mathcal{W}) = \{\mathbf{x} \in \mathcal{B}(n) \,:\, w(\mathbf{x}) \in \mathcal{W} \text{ and } r(\mathbf{x}) \leq r\}. \tag{2.8}$$

We denote its cardinality by $B_r(n, \mathcal{W})$.

To clarify this we will consider the following example:

Example 2.4. Consider the set $\mathcal{B}_1(3, \{1, 2\})$ then both $\mathbf{x}_1 = [1, 0, 2]$ and $\mathbf{x}_2 = [2, 1, 3]$ are in this set since $w(\mathbf{x}_1) = 1$ and $w(\mathbf{x}_2) = 2$. But $\mathbf{x}_3 = [1, 0, 1]$ is not in the set because $w(\mathbf{x}_3) = 0$.

As mentioned in the previous section, errors can occur when we want to store data in DNA. By imposing the GC-weight and the maximum run-length we lower the chances of errors. But it is still possible that errors occur. Therefore we want a DNA-code to also satisfy a minimum Hamming distance. The minimum distance between words in a code is an important concept to detect errors and even correct them. First we will give the definition of the distance between two words, after that the definition of the minimum distance of a code is defined.

Definition 2.7. Let $\mathbf{x}$ and $\mathbf{y}$ be two words of the same length. We define the **Hamming distance between two words** as the number of positions at which the corresponding symbols are different i.e.

$$d(\mathbf{x}, \mathbf{y}) = |\{i : x_i \neq y_i\}|. \tag{2.9}$$

Example 2.5. Consider two words in $\mathcal{B}(5)$:
$\mathbf{x} = [1, 0, 2, 3, 1]$,
$\mathbf{y} = [1, 2, 2, 1, 3]$.
We see that these words have a Hamming distance of 3.

Definition 2.8. We define the **Hamming distance of a DNA-code** $\mathcal{C}$, denoted by $d(\mathcal{C})$, as the smallest distance between any two different words $\mathbf{x}$ and $\mathbf{y}$ in $\mathcal{C}$:

$$d(\mathcal{C}) = \min\{d(\mathbf{x}, \mathbf{y}) \mid \mathbf{x}, \mathbf{y} \in \mathcal{C} \,:\, \mathbf{x} \neq \mathbf{y}\}. \tag{2.10}$$

If a DNA-code satisfies Hamming distance d then any two DNA-words in the DNA-code have a distance greater than or equal to d.

Definition 2.9. We define a **DNA-d** code $\mathcal{C}_d$ as a code which has minimum Hamming distance d.

Definition 2.10. We define the size of the largest DNA-d code as

$$B_r(n, \mathcal{W}, d) = \max\{|\mathcal{C}| \,:\, \mathcal{C} \subseteq \mathcal{B}_r(n, \mathcal{W}),\, d(\mathcal{C}) \geq d\}. \tag{2.11}$$

The minimum distance is an important concept because it limits the number of words that can be in the code. Due to this limitation, all words in the code have a certain distance from each other, which means that there is a minimum number for the number of symbols that words must differ from each other. By using this, errors can be detected and even corrected. For example if we consider a code with minimum Hamming distance 2 then if one error occurs the word is not in the code so the error can be detected. We can generalize this: In a code with minimum Hamming distance d, up to $d - 1$ errors are guaranteed to be detected. Additionally, up to $\lfloor \frac{d-1}{2} \rfloor$ errors can be corrected.

3

The maximum size of DNA-n codes

In the search for DNA-codes of maximum size, we can consider codes with an extreme value for the minimum Hamming distance. Namely codes where the minimum Hamming distance is equal to the length n. In [3] this case is also considered for various values of w. In this chapter we will consider the case where we have a set of GC-weights $\mathcal{W}$ and the words in the code need to satisfy one of the GC-weights in this set.

Throughout this chapter we will represent the DNA-codes in matrix form. We will do this as follows. The rows of the matrices represent the DNA-words and the columns represent the symbols at the positions in the words. This means that $c_{i,j}$ is the j[th] symbol of word $\mathbf{x}_i$. By representing the DNA-code in matrix form, we number the words of the code. We say that the first row of the matrix represents the first word of the DNA-code.

3.1. An upper bound for DNA-n codes

We first obtain an upper bound for the maximum size of a DNA-n code.

Theorem 3.1. $B_r(n, \mathcal{W}, n) \leq 4$ for all $n, \mathcal{W}, r$.

Proof. Suppose we have the largest possible code $\mathcal{C}_n$, with $\mathcal{C}_n \subseteq \mathcal{B}_r(n, \mathcal{W})$ and $d(\mathcal{C}_n) \geq n$. From Equation 2.9, we observe that for all $j \in \{1, \dots, n\}$ the constraint $d = n$ requires

$$c_{i,j} \neq c_{l,j} \; \forall i, l \in \{1, \dots, |\mathcal{C}_n|\} \text{ with } i \neq l. \tag{3.1}$$

Because the DNA-words are words in the quaternary alphabet, we have four different symbols, $\{0, 1, 2, 3\}$. Therefore Equation 3.1 implies $|\mathcal{C}_n| \leq 4$. Thus we conclude:

$$B_r(n, \mathcal{W}, n) \leq 4. \tag{3.2}$$

□

3.2. The case where the maximum size of a DNA-n code is 4

In the previous section we found an upper bound for the maximum size of a DNA-n code, now the question remains if we can find valid DNA-codes such that we can conclude $B_r(n, \mathcal{W}, n) \geq 4$. From Equation 2.8 we see that DNA-codes which satisfy $r = 1$, satisfy all $r > 1$. So if we find a DNA-code of size 4, satisfying $r = 1$, this DNA-code is also a valid DNA-code for $r > 1$.

Theorem 3.2. $B_r(n, \mathcal{W}, n) = 4$ if and only if there exists $w_1, w_2, w_3, w_4 \in \mathcal{W}$ such that $w_1 + w_2 + w_3 + w_4 = 2n$.

Proof. Suppose we have $w_1, w_2, w_3, w_4 \in \mathcal{W}$ such that $w_1 + w_2 + w_3 + w_4 = 2n$ and suppose that without loss of generality we have $0 \leq w_1 \leq w_2 \leq w_3 \leq w_4 \leq n$. We will show that we can find a valid DNA code $\mathcal{C} \subseteq \mathcal{B}_r(n, \mathcal{W})$ with $d(\mathcal{C}) \geq n$ consisting of 4 words, and therefore that this is a lower bound for

$B_r(n, \mathcal{W}, n)$. From Theorem 3.1 we know that we have an upper bound of 4, so if we can find such a code we know that this is the largest possible size.
As said before we will represent the DNA code in matrix form as follows:

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

Table 3.1: The DNA-n code consisting of four words.

As you see in Table 3.1 the rows represent the DNA words. Because we consider the case $d = n$, the words need to differ from each other at each position. This means that in every column each element from $\{0, 1, 2, 3\}$ needs to appear exactly once.

We can find a code satisfying the constraints and consisting of four words, by splitting the code into four parts, see Figure 3.1. In this figure the boxes are numbered, the number of the box is underlined.

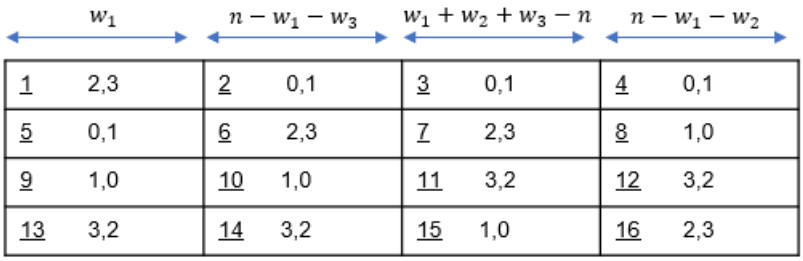


Figure 3.1: Schematic overview of how the DNA-n code consisting of 4 words looks like.

The first part consists of w_1 symbols, the second part of $n - w_1 - w_3$ symbols, the third part of $w_1 + w_2 + w_3 - n$ symbols and the fourth part consists of $n - w_1 - w_2$ symbols. In this way every word consists of n symbols, and thus has a length of n.

- For the first word we see that the symbols in the first part, that consists of w_1 symbols, alternate between 2 and 3. Then the symbols in the three other parts are alternately 0 and 1. In this way the first word has a GC-weight of w_1.
- For the second word the first w_1 symbols alternate between 0 and 1. The following $n - w_1 - w_3$ and then also the following $w_1 + w_2 + w_3 - n$ symbols are alternately 2 and 3. Lastly the last $n - w_1 - w_2$ symbols of this word are alternately 0 and 1 again. We see that this word has $n - w_1 - w_3 + w_1 + w_2 + w_3 - n = w_2$ elements from $\{2, 3\}$ and thus this second word has a GC-weight of w_2.
- The third word of this DNA-code has the first w_1 symbols alternating between 0 and 1. Also, the following $n - w_1 - w_3$ elements are alternating between 0 and 1. The following $w_1 + w_2 + w_3 - n$ and the last $n - w_1 - w_2$ elements alternate between 2 and 3. So we see that this word has a GC-weight of $w_1 + w_2 + w_3 - n + n - w_1 - w_2 = w_3$.
- The last and fourth word of this code has the first w_1 and the following $n - w_1 - w_3$ elements alternating between 2 and 3. Then the following $w_1 + w_2 + w_3 - n$ symbols alternate between 0 and 1. And the last $n - w_1 - w_2$ symbols of this fourth word alternate between 2 and 3. We see that this word has a GC-weight of $w_1 + n - w_1 - w_3 + n - w_1 - w_2 = 2n - w_1 - w_2 - w_3 = w_4$.

From this we see that we can make codes where the four words have GC-weight w_1, w_2, w_3 and w_4. To make sure that the maximum run length constraint $r = 1$ is satisfied and to make sure all the words from the code have Hamming distance $d = n$ from each other we need to determine with which of the two symbols the alternation in each box starts.

For box 1 and 2 this choice is free and by these two choices also for the rest of the boxes it is determined with which symbol the alternation starts. The figure below shows the order in which you know how to fill in the boxes.

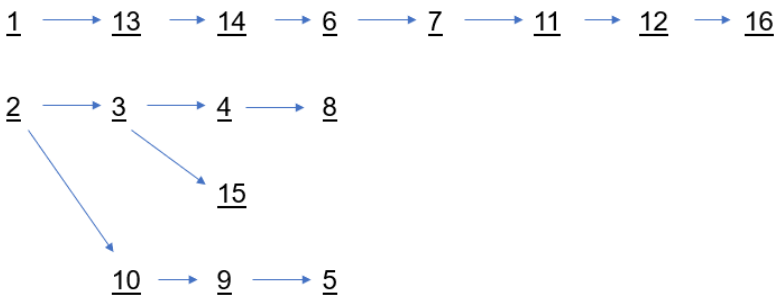


Figure 3.2: Schematic overview of the order in which you can fill in the boxes to create a valid DNA-n code.

To give an example: if you start the alternation in box 1 with 2, then because we need $d = n$ the alternation in box 13 has to start with 3, then because we need $r = 1$ we also know with which symbol we need to start the alternation in box 14 and in this way you can fill in the rest of the boxes.
In addition, it is also possible that one of the four parts has a length of 0. In that case, this part will be omitted, but still all the boxes can be filled through the other boxes. The order in which you can fill in the boxes will only change a little.

So we found that we can make a DNA-code satisfying $r = 1$ consisting of four words that have a Hamming distance of $d = n$ from each other, and thus we can conclude $B_r(n, \mathcal{W}, n) \geq 4$ if there exists $w_1, w_2, w_3, w_4 \in \mathcal{W}$ such that $w_1 + w_2 + w_3 + w_4 = 2n$.

Hence together with the upper bound we found in Theorem 3.1 we conclude:
If there exists $w_1, w_2, w_3, w_4 \in \mathcal{W}$ such that $w_1 + w_2 + w_3 + w_4 = 2n$ then $B_r(n, \mathcal{W}, n) = 4$.

Now suppose we have $B_r(n, \mathcal{W}, n) = 4$ so there exists a code $\mathcal{C}$ with 4 words. Then because we have $d = n$ the words need to differ from each other in each position. Because we have a code with four words, this means that in every column each element from $\{0, 1, 2, 3\}$ appears exactly once. This means that every column contains 2 elements from $\{2, 3\}$. We have words of length n so the GC-weight of all the words together is equal to $2n$. So if we let w_1 be the GC-weight of $\mathbf{c}_1 \in \mathcal{C}$, w_2 be the GC-weight of $\mathbf{c}_2 \in \mathcal{C}$, etc. Then we have $w_1, w_2, w_3, w_4 \in \mathcal{W}$ and $w_1 + w_2 + w_3 + w_4 = 2n$.
Hence we conclude $B_r(n, \mathcal{W}, n) = 4$ if and only if $\exists w_1, w_2, w_3, w_4 \in \mathcal{W}$ such that $w_1 + w_2 + w_3 + w_4 = 2n$. □

3.3. DNA-n codes with maximum size smaller then 4

In Section 3.2 we proved the case when $B_r(n, \mathcal{W}, n) = 4$. Now we will consider the case when there is no $w_1, w_2, w_3, w_4 \in \mathcal{W}$ with $w_1 + w_2 + w_3 + w_4 = 2n$ and thus the cases where $B_r(n, \mathcal{W}, n) < 4$.

Theorem 3.3. $B_r(n, w, n) = 3$ if and only if there does not exist $w_i, w_j, w_k, w_l \in \mathcal{W}$ such that $w_i + w_j + w_k + w_l = 2n$ but we have $w_1, w_2, w_3 \in \mathcal{W}$ such that $n \leq w_1 + w_2 + w_3 \leq 2n$.

Proof. We will first show that there is a valid code satisfying the constraints and that consist of three words, and thus that we can conclude that we found a lower bound for $B_r(n, w, n)$.

Suppose have $w_1, w_2, w_3 \in \mathcal{W}$ such that $n \leq w_1 + w_2 + w_3 \leq 2n$. Furthermore suppose without loss of generality that we have $0 \leq w_1 \leq w_2 \leq w_3 \leq n$. We will consider the code in matrix form again:

$c_{1,1}$	…	$c_{1,n}$
$c_{2,1}$	…	$c_{2,n}$
$c_{3,1}$	…	$c_{3,n}$

Let the first row of the matrix, so the first word from the code be as followed:

Let

- $c_{1,1}, \dots, c_{1,w_1}$ alternately be elements from $\{2,3\}$, starting with $c_{1,1} = 2$.
- $c_{1,w_1+1}, \dots, c_{1,n}$ alternately be 0 and 1 staring with 0.

Then we consider three cases, the case where $w_1 + w_2 < n$ and $w_2 + w_3 < n$, the case where $w_1 + w_2 < n$ and $w_2 + w_3 > n$ and the case where $w_1 + w_2 > n$. We consider these three cases because it is not possible that we have $w_1 + w_2 = n$ or $w_2 + w_3 = n$. If this would be the case then we would have that $w_1 + w_2 + w_1 + w_2 = 2n$ and $w_1, w_2 \in \mathcal{W}$ or $w_2 + w_3 + w_2 + w_3 = 2n$ and $w_2, w_3 \in \mathcal{W}$ and then from Theorem 3.2 we would know that $B_r(n, w, n) = 4$.

- Suppose we have the case where $w_1 + w_2 < n$ and $w_2 + w_3 < n$ in this case let the second and third row be as follows:

  Let $c_{2,1}, \dots, c_{2,w_1}$ be alternately 0 ad 1. If w_1 is even start with 0 and if w_1 is odd start with 1.
  Let $c_{2,w_1+1}, \dots, c_{2,w_1+w_2}$ be alternately 2 and 3. If w_1 even start with 2 and if w_1 odd start with 3.
  And let $c_{2,w_1+w_2+1}, \dots, c_{2,n}$ be alternately 0 and 1 again, starting with 1 if w_2 even and starting with 0 if w_2 is odd.

  For the third row, let $c_{3,1}, \dots, c_{3,w_3-n+w_1+w_3}$ be alternately 2 and 3 starting with 3.
  Let $c_{3,w_3-n+w_1+w_3+1}, \dots, c_{3,w_1+w_2}$ be alternately 0 and 1. If $c_{2,w_3-n+w_1+w_3+1}$ is 0 start with 1, otherwise start with 0.
  And let $c_{3,w_1+w_2+1}, \dots, c_{3,n}$ be alternately 2 and 3 again. We start with 2 if $w_1 + w_2$ is odd and with 3 if $w_1 + w_2$ is even.

$w_3 - n + w_1 + w_2$	$n - w_2 - w_3$	w_2	$n - w_1 - w_2$
$2,3$	$2,3$	$0,1$	$0,1$
$0,1$	$0,1$	$2,3$	$1,0$
$3,2$	$1,0$	$1,0$	$3,2$

Figure 3.3: Schematic overview of how the DNA-n code consisting of 3 words looks like when $w_1 + w_2 < n$ and $w_2 + w_3 < n$.

- Now suppose that we have $w_1 + w_2 < n$ and $w_2 + w_3 > n$ in this case let the second and third row be as follows:

  Let $c_{2,1}, \dots, c_{2,w_1}$ be alternately 0 ad 1. If w_1 is even start with 0 and if w_1 is odd start with 1.
  Let $c_{2,w_1+1}, \dots, c_{2,w_1+w_2}$ be alternately 2 and 3. If w_1 even start with 2 and if w_1 odd start with 3.
  And let $c_{2,w_1+w_2+1}, \dots, c_{2,n}$ be alternately 0 and 1 again, starting with 1 if w_2 even and starting with 0 if w_2 is odd.

  For the third row, let $c_{3,1}, \dots, c_{3,w_1+w_2+w_3-n}$ be alternately 2 and 3 starting with 3.
  Let $c_{3,w_1+w_2+w_3-n+1}, \dots, c_{3,w_1+w_2}$ be alternately 0 and 1. If $c_{1,w_1+w_2+w_3-n+1}$ is 0 start with 1, otherwise start with 0.
  And let $c_{3,w_1+w_2+1}, \dots, c_{3,n}$ be alternately 2 and 3 again. We start with 2 if $w_1 + w_2$ is odd and with 3 if $w_1 + w_2$ is even.

w_1	$w_2 + w_3 - n$	$n - w_3$	$n - w_1 - w_2$
2, 3	0,1	0, 1	0, 1
0,1	2,3	2, 3	1,0
3,2	3,2	1,0	3,2

Figure 3.4: Schematic overview of how the DNA-n code consisting of 3 words looks like when $w_1 + w_2 < n$ and $w_2 + w_3 > n$.

- Suppose now we have $w_1 + w_2 > n$, then we let the second and third row be as follows:

  Let $c_{2,1}, \dots c_{2,w_1+w_2-n}$ be alternately 2 and 3 starting with 3.
  Let $c_{2,w_1+w_2-n+1}, \dots, c_{2,w_1}$ be alternately 0 and 1, starting with 0.
  And let $c_{2,w_1+1}, \dots, c_{2,n}$ be alternately 2 and 3 again starting with 2 if w_1 is even, and staring with 3 if w_1 is odd.

  For the third row let $c_{3,1}, \dots, c_{3,w_1+w_2-n}$ be alternately 0 and 1 starting with 0.
  Let $c_{3,w_1+w_2-n+1}, \dots, c_{3,w_1+w_2+w_3-n}$ be alternately 2 ad 3 starting with 2 if $w_1 + w_2 - n$ is odd and starting with 3 if $w_1 + w_2 - n$ is even.
  And let $c_{3,w_1+w_2+w_3-n+1} \dots, c_{3,n}$ be alternately 0 and 1 again, starting with 0 if $c_{1,w_1+w_2-n+w_3+1} = 1$ and 1 if $c_{1,w_1+w_2-n+w_3+1} = 0$.

$w_1 + w_2 - n$	$n - w_2$	$w_2 + w_3 - n$	$2n - w_1 - w_2 - w_3$
2, 3	2, 3	0, 1	0, 1
3, 2	0, 1	2, 3	2, 3
0, 1	3, 2	3, 2	1, 0

Figure 3.5: Schematic overview of how the DNA-n code consisting of 3 words looks like when $w_1 + w_2 > n$.

These are valid codes in $\mathcal{B}_r(n, \mathcal{W})$ where the distance between all the words is n and thus we can conclude that if $w_1, w_2, w_3 \in \mathcal{W}$ with $n \leq w_1 + w_2 + w_3 \leq 2n$ then $B_r(n, w, n) \geq 3$.
Now from Theorem 3.2 we know that if we have a DNA code $\mathcal{C} \subseteq \mathcal{B}_r(n, \mathcal{W})$ and $d(\mathcal{C}) \geq n$ with $|\mathcal{C}| = 4$ then we have $w_1, w_2, w_3, w_4 \in \mathcal{W}$. Thus we know that if there does not exist $w_i, w_j, w_k, w_l \in \mathcal{W}$ such that $w_i + w_j + w_k + w_l = 2n$ but we have $w_1, w_2, w_3 \in \mathcal{W}$ such that $n \leq w_1 + w_2 + w_3 \leq 2n$ then $B_r(n, w, n) \leq 3$. And thus $B_r(n, w, n) = 3$.

Now suppose we have $B_r(n, w, n) = 3$. Then let w_1 be the weight of $\mathbf{c}_1$, w_2 the GC-weight of $\mathbf{c}_2$ and w_3 the GC-weight of $\mathbf{c}_3$. Then the total weight of these words is $w_1 + w_2 + w_3$. Then because we have $d = n$ we need that for all $j \in \{1, \dots, n\}$ we have $c_{i,j} \neq c_{l,j}$ for $i, l \in 1, 2, 3$ and $i \neq l$. This means that every column has one or two symbols from $\{2, 3\}$, and we have words of length n and thus we need $n \leq w_1 + w_2 + w_3 \leq 2n$. So together with Theorem 3.2 we conclude that if $B_r(n, w, n) = 3$ then there does not exist $w_i, w_j, w_k, w_l \in \mathcal{W}$ such that $w_i + w_j + w_k + w_l = 2n$ but we have $w_1, w_2, w_3 \in \mathcal{W}$ such that $n \leq w_1 + w_2 + w_3 \leq 2n$.

Hence we conclude $B_r(n, w, n) = 3$ if and only if there does not exist $w_i, w_j, w_k, w_l \in \mathcal{W}$ such that $w_i + w_j + w_k + w_l = 2n$ but we have $w_1, w_2, w_3 \in \mathcal{W}$ such that $n \leq w_1 + w_2 + w_3 \leq 2n$.

□

Theorem 3.4. If we have $\mathcal{W}$ such that there is no $w_1, w_2, w_3 \in \mathcal{W}$ with $n \leq w_1 + w_2 + w_3 \leq 2n$ then $B_r(n, \mathcal{W}, n) = 2$.

Proof. We will first show that we can find a valid DNA-code satisfying the constraints and consisting of two words. Consider the following code.

Let the first word of this code be the word starting with $c_{1,1}, \dots, c_{1,w_1}$ being alternately 2 and 3 starting with $c_{1,1} = 2$. And let $c_{1,w_1+1}, \dots, c_{1,n}$ from right to left be alternately 0 and 1, starting with 0 being the rightmost bit.

For the second word let $c_{2,1}, \dots, c_{2,w_2}$ be 2 and 3 alternately starting with $c_{2,1} = 3$. And let $c_{2,w_2+1}, \dots, c2, n$ from right to left be alternately 0 and 1 starting with 1.
Then this is a valid code for the case where we there is no $w_1, w_2, w_3 \in \mathcal{W}$ with $n \leq w_1 + w_2 + w_3 \leq 2n$, so we can conclude that $B_r(n, \mathcal{W}, n) \geq 2$.
Now suppose there does not exist $w_1, w_2, w_3 \in \mathcal{W}$ with $n \leq w_1 + w_2 + w_3 \leq 2n$ then we also can not have $w_1, w_2, w_3, w_4 \in \mathcal{W}$ such that $w_1 + w_2 + w_3 + w_4 = 2n$. Thus from Theorem 3.2 and 3.3 we know

$B_r(n,w,n) \neq 4$ and $B_r(n,w,n) \neq 3$ respectively. And thus together with Theorem 3.1 which says that $B_r(n,w,n) \leq 4$ for all $n, \mathcal{W}, r$ we can conclude that $B_r(n,w,n) \leq 2$. Hence we can conclude that if there does not exist $w_1, w_2, w_3 \in \mathcal{W}$ with $n \leq w_1 + w_2 + w_3 \leq 2n$ then $B_r(n,\mathcal{W},n) = 2$.

□

3.4. Overview

In conclusion, Theorem 3.2, 3.3 and 3.4 show:

$$\forall r, n \geq 1, B_r(n,\mathcal{W},n) = \begin{cases} 4 & \text{If there exists } w_1, w_2, w_3, w_4 \in \mathcal{W} \text{ such that } w_1 + w_2 + w_3 + w_4 = 2n \\ 2 & \text{If we have } \mathcal{W} \text{ such that there is no } w_1, w_2, w_3 \in \mathcal{W} \text{ with } n \leq w_1 + w_2 + w_3 \leq 2n \\ 3 & \text{otherwise} \end{cases} \tag{3.3}$$

To illustrate this we give an example.

Example 3.1. In this example we will determine the maximum size of DNA-n codes where the words have length 6 for different sets of GC-weights. So we will determine $B_r(6,\mathcal{W},6)$ for different sets $\mathcal{W}$.

- Suppose we have $\mathcal{W} = \{3\}$ then $3+3+3+3 = 12 = 2*6$ and so we have $B_r(6,\mathcal{W},6) = 4$. An example of such a valid code is:

$$\{020202, 131313, 202020, 313131\}.$$

- Suppose we have $\mathcal{W} = \{2,4\}$ then we have $2+4+2+4 = 12 = 2*6$ and thus again $B_r(6,\mathcal{W},6) = 4$. An example of such a code of maximum size would be:

$$\{230101, 012323, 101032, 323210\}.$$

- Suppose $\mathcal{W} = \{4\}$ then we have $4+4+4+4 = 16 \neq 2*6$ but $4+4+4 = 12$ so $6 \leq 12 \leq 2*6$ and thus we have $B_r(6,\mathcal{W},6) = 3$. An example of a code of maximum size would be:

$$\{232301, 320123, 013232\}.$$

- Suppose $\mathcal{W} = \{1\}$ we have that $1+1+1 = 3 \leq 6$ and thus we have that $B_r(6,\mathcal{W},6) = 2$. An example of such a code with maximum size is:

$$\{201010, 310101\}$$

4

The maximum size of a DNA-code with $d = n - 1$

In this chapter, we will consider codes of length n that have a minimum Hamming distance of $d = n - 1$ and run-length $r = 1$. Furthermore, we will consider DNA-codes where the weights of the DNA-words are around half the length of the code words. This means for even-length codes that $\frac{n}{2} - 1, \frac{n}{2}, \frac{n}{2} + 1 \in \mathcal{W}$ and for odd-length codes that $\frac{n-1}{2}, \frac{n+1}{2} \in \mathcal{W}$.

Throughout this chapter, we will again represent the DNA codes in matrix form.

For several proofs in this chapter, we use the idea of the proof of the Plotkin bound [11]. The Plotkin bound is an upper bound for the maximum possible number of code words in binary codes of given length n and given minimum Hamming distance d. We let the expression $A(n, d)$ represent this maximum number of possible code words. Since we use the proof of Plotkin's theorem in this chapter we will give the theorem and the proof.

Theorem 4.1. If $2d > n$ then $A(n, d) \leq \frac{2d}{2d-1}$.

Proof. Let K be any code consisting of A words of length n with minimum Hamming distance d. The bound is proved by bounding the quantity

$$N = \sum_{(\mathbf{x},\mathbf{y}) \in K^2, \mathbf{x} \neq \mathbf{y}} d(\mathbf{x}, \mathbf{y})$$

in two different ways.

On the one hand, there are A choices of $\mathbf{x}$ and for each such choice there are $A - 1$ choices for $\mathbf{y}$. The code has a minimum Hamming distance of d and therefore we know $d(\mathbf{x}, \mathbf{y}) \geq d$ for all $\mathbf{x}$ and $\mathbf{y}$ ($\mathbf{x} \neq \mathbf{y}$). Hence it follows that

$$N \geq A(A - 1)d.$$

On the other hand, consider the code K in matrix form, such that we have an A x n matrix. Let m_i be the number of elements that are 0 in the ith column. Then there are $(A - m_i)$ elements 1 in the first column. Because we have $d(\mathbf{x}, \mathbf{y}) = d(\mathbf{y}, \mathbf{x})$, we have that every choice of a 0 and a 1 in the same comlum contributes 2 tot the sum. Therefore we have:

$$N = \sum_{i=1}^{n} 2m_i(A - m_i)$$

Now we consider two cases, the case where A is even and the case where A is odd:
Case 1): Consider $A = 2m$, so A is even. In that case the sum $\sum_{i=1}^{n} 2m_i(A - m_i)$ is maximum if we have $m_i = \frac{A}{2}$ for all i, we then have

$$N \leq \frac{1}{2}nA^2.$$

Combining this inequality with $N \geq A(A-1)d$,

$$2(A-1)d \leq An$$

and

$$(2d - n)A \leq 2d.$$

And since we had $2d > n$ we find:

$$A \leq \frac{2d}{2d-n}.$$

Case 2): Now consider $A = 2m - 1$, so A is odd. In this case the sum $\sum_{i=1}^{n} 2m_i(A - m_i)$ is maximum if we have $m_i = \frac{A-1}{2}$ for all i, we then have

$$N \leq \frac{1}{2}n(A^2 - 1).$$

Continuing as in Case 1) we see that

$$2A(A-1)d \leq A^2n - n < A^2n$$

and thus we find:

$$A < \frac{2d}{2d-n}.$$

□

Definition 4.1. We define $C_{j,k} = \{\mathbf{c}_i : c_{i,j} = k\}$ where $k = 0, 1, 2, 3$. This is the set of words that have the same symbol k at position j.

Example 4.1. We consider the following code $\mathcal{C}$ in matrix form.

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

Table 4.1: A DNA-code with $n = 4$.

Then $C_{1,0} = \{\mathbf{c}_1, \mathbf{c}_4\}$, $C_{1,1} = \emptyset$, $C_{1,2} = \{\mathbf{c}_3\}$ and $C_{1,3} = \{\mathbf{c}_2\}$.

We first obtain an upper bound for the maximum size of DNA-words with $d = n - 1$

Theorem 4.2. $B_1(n, \mathcal{W}, n-1) \leq 12$ for all $n, \mathcal{W}$.

Proof. Suppose we have a DNA-code $\mathcal{C}$ with at least 13 DNA-words, satisfying the constraints. Then $C_{1,0}$ is the set of words of the code $\mathcal{C}$ that have 0 as their first symbol, $C_{1,1}$ is the set of words of the code $\mathcal{C}$ that have 1 as their first symbol, etc. Because $|\mathcal{C}| \geq 13$ we have

$$|C_{1,l}| \geq 4 \text{ for some } l \in \{0, 1, 2, 3\}.$$

Now suppose without loss of generality that we have four DNA-words starting with 0, so suppose

$$C_{1,0} = \{01 \ldots, 02 \ldots, 03 \ldots, 0k \ldots\}.$$

In case $k = 1, 2, 3$ the constraint $d = n - 1$ is no longer satisfied. In case $k = 0$ the constraint for the run-length, $r = 1$, is no longer satisfied. From this we can conclude that for each symbol $k \in \{0, 1, 2, 3\}$ the DNA-code $\mathcal{C}$ is not a valid code with $\mathcal{C} \subseteq \mathcal{B}_1(n, \mathcal{W})$ and $d(\mathcal{C}) \geq d$. So we need $B_1(n, \mathcal{W}, n-1) \leq 12$.

□

Remark 1. Note that for each position, we can have a maximum of three valid DNA-words with the same symbol at this position. If there is another DNA-word with this same symbol at the same position, again the run-length constraint, $r = 1$, is not satisfied anymore, or the distance between this word and one of the other words is less then $d = n - 1$. Hence we have that for all $j \in \{1, \dots, n\}$

$$|C_{j,k}| \leq 3. \tag{4.1}$$

From Theorem 4.2 we see that we found an upper bound for the maximum size of codes with $d = n - 1$. For each length n, we will now try to find lower bounds by creating valid DNA codes and we will try if it is possible to lower the upper bound, in order to find the maximum size.

Theorem 4.3. $B_1(3, \mathcal{W}, 2) = 12$ if $\{1, 2\} \subseteq \mathcal{W}$.

Proof. To prove that $B_1(3, \mathcal{W}, 2) = 12$ if $\{1, 2\} \subseteq \mathcal{W}$ we are looking for upper and lower bounds for the largest DNA-code $\mathcal{C}$ such that $\mathcal{C} \subseteq \mathcal{B}_1(3, \mathcal{W})$ where $\{1, 2\} \subseteq \mathcal{W}$ and $d(\mathcal{C}) \geq 2$.
Theorem 4.2 gave us an upper bound for the size of a DNA code with $r = 1$ and $d = n - 1$. Therefore we know $B_1(3, \mathcal{W}, 2) \leq 12$ for all $\mathcal{W}$.
Now consider the following valid code $\mathcal{C}$:

$$\mathcal{C} = \{012, 023, 031, 103, 120, 132, 201, 213, 230, 302, 310, 321\}.$$

This code is obtained from the code used in [3] to prove $B_1(4, [2], 3) = 12$. We see that if we use this code and remove the last symbol from all DNA-words in this code , we obtain words of length 3, weight 1 or 2 and with run-length still equal to 1. Because the code of length 4 had a minimum Hamming distance of 3, when removing one bit of every word we get a minimum Hamming distance of at least 2. And thus the DNA-code $\mathcal{C}$ is a valid code with $\mathcal{C} \subseteq \mathcal{B}_1(3, \mathcal{W})$ where $\{1, 2\} \subseteq \mathcal{W}$ and $d(\mathcal{C}) \geq 2$.

We see that the size of this code is 12. So we found a lower bound for largest DNA-code, $B_1(3, \mathcal{W}, 2) \geq 12$ if $\{1, 2\} \subseteq \mathcal{W}$.
Hence we can conclude $B_1(3, \mathcal{W}, 2) = 12$ if $\{1, 2\} \subseteq \mathcal{W}$. □

Theorem 4.4. $B_1(4, \mathcal{W}, 3) = 12$ if $\{2\} \subseteq \mathcal{W}$.

Proof. To prove $B_1(4, \mathcal{W}, 3) = 12$ if $\{2\} \subseteq \mathcal{W}$ we will show that we can find an upper and lower bound for $B_1(4, \mathcal{W}, 3)$ with $2 \in \mathcal{W}$ which are both equal to 12.
Theorem 4.2 gave us an upper bound for the size of a DNA-code with $r = 1$ and $d = n - 1$. Therefore we know that $B_1(4, \mathcal{W}, 3) \leq 12$for all $\mathcal{W}$.
In [3] it was proved that $B_1(4, \{2\}, 3) = 12$, to prove this the following code was found to determine the lower bound:

$$\mathcal{C} = \{0123, 0231, 0312, 1032, 1203, 1320, 2013, 2130, 2301, 3021, 3102, 3210\}.$$

Note that $\mathcal{C}$ is also a valid code for $C \subseteq \mathcal{B}_1(4, \mathcal{W})$ with $2 \in \mathcal{W}$ and $d(\mathcal{C}) \geq 3$. And thus the size of this DNA-code is also a lower bound for $B_1(4, \mathcal{W}, 3)$ with $\{2\} \subseteq \mathcal{W}$, so $B_1(4, \mathcal{W}, 3) \geq 12$ if $\{2\} \subseteq \mathcal{W}$.
Hence we can conclude

$$B_1(4, \mathcal{W}, 3) = 12 \text{ if } \{2\} \subseteq \mathcal{W}.$$

□

Theorem 4.5. $10 \leq B_1(5, \mathcal{W}, 4) \leq 12$ if $\{2, 3\} \subseteq \mathcal{W}$.

Proof. From Theorem 4.2 we now that $B_1(5, \mathcal{W}, 4) \leq 12$ and thus we found an upper bound.
For the lower bound we consider the following code:

$$\mathcal{C} = \{01230, 02313, 03121, 10320, 12031, 13203, 21301, 23010, 31023, 32102\}.$$

This is a valid code in $\mathcal{B}_1(5, \mathcal{W})$ with $\{2, 3\} \subseteq \mathcal{W}$ and with $d(\mathcal{C}) \geq 4$. We note $|\mathcal{C}| = 10$. Therefore we have

$$B_1(5, \mathcal{W}, 4) \geq 10 \text{ for all } \mathcal{W} \text{ where } 2, 3 \in \mathcal{W}.$$

Hence we can conclude

$$10 \leq B_1(5, \mathcal{W}, 4) \leq 12 \text{ if } \{2, 3\} \subseteq \mathcal{W}.$$

□

We will now present some definitions, that we use when finding upper bounds for codes with length $n > 5$.

Definition 4.2. For $i \in \{1, \dots, |\mathcal{C}|\}$ we define $G_i = \{C_{j,k} : \mathbf{c}_i \in C_{j,k}$ for $j \in \{1, \dots, n\}$ and $k \in \{0, 1, 2, 3\}\}$. This is the set of sets $C_{j,k}$ where the DNA-word $\mathbf{c}_i$ belongs to.

Example 4.2. If we consider the code from Example 4.1 again we have

$$G_1 = \{C_{1,0}, C_{2,1}, C_{3,2}, C_{4,3}\}.$$

Note that in this chapter we consider codes where $d = n - 1$ therefore we have

$$|G_i \cap G_l| \leq 1 \forall i, l \in \{1, \dots, |\mathcal{C}|\} \text{ and } i \neq l. \tag{4.2}$$

Because if for some i, l we have $|G_i \cap G_l| \geq 2$ then $\mathbf{c}_i$ and $\mathbf{c}_l$ have on at least 2 positions the same symbol but then $d = n - 1$ is not satisfied anymore.

Definition 4.3. We define the set $P_j(m) = \{C_{j,k} : |C_{j,k}| = m$ for $k \in \{0, 1, 2, 3\}\}$, as the set containing all the sets $C_{j,k}$ that have the cardinality m.

Example 4.3. Consider the DNA-code from Table 4.1 again. Then we have:

$$P_1(1) = \{C_{1,2}, C_{1,3}\},$$

because $C_{1,0} = \{\mathbf{c}_1, \mathbf{c}_4\}$, $C_{1,1} = \emptyset$, $C_{1,2} = \{\mathbf{c}_3\}$ and $C_{1,3} = \{\mathbf{c}_2\}$, so there are two sets, $C_{1,2}, C_{1,3}$, that have cardinality 1 for $j = 1$.

Theorem 4.6. $B_1(6, \mathcal{W}, 5) \leq 9$ for all $\mathcal{W}$.

Proof. To proof that $B_1(6, \mathcal{W}, 5) \leq 9$ for all $\mathcal{W}$ we will use the idea of the proof of Plotkin that we gave in Theorem 4.1.

Suppose we have a code $\mathcal{C} \subseteq \mathcal{B}_1(6, \mathcal{W})$ with $d(\mathcal{C}) \geq 5$ and with $|\mathcal{C}| = 10$. Then because we have $|\mathcal{C}| = 10$, we need that for all $j \in \{1, \dots 6\}$ we have $\sum_{k=0}^{3} |C_{j,k}| = 10$. Therefore we need that for all j we have

$$|P_j(3)| = 3 \text{ and } |P_j(1)| = 1, \text{ or}$$
$$|P_j(3)| = 2 \text{ and } |P_j(2)| = 2.$$

We need this because for all j we have at most four different sets $C_{j,k}$ and for each such set we have $|C_{j,k}| \leq 3$.

Following the method of Plotkin [11] we will calculate the sum

$$N = \sum_{c_i \in \mathcal{C}} \sum_{c_l \in \mathcal{C}} d(c_i, c_l) \tag{4.3}$$

in two ways.

- Since $d(c_i, c_j) \geq d \; \forall c_i, c_l \in \mathcal{C}$ with $c_i \neq c_l$, we have

  $$N \geq |\mathcal{C}|(|\mathcal{C}| - 1)d = 10 * (10 - 1) * 5 = 450.$$

- Now if we consider the DNA-code $\mathcal{C}$ in matrix form and we look at the columns of the matrix we see that the total distance of column is depending on the $C_{j,k}$. Because if $c_1, c_2 \in C_{j,k}$ for some j and some k then the distance between these two DNA-words in the column j is 0.

  Therefore if for a column we have $|P_j(3)| = 3$ and $|P_j(1)| = 1$ then this column contributes

  $$\sum_{m=0}^{3} m * |P_j(m)| * (|\mathcal{C}| - m) = 1 * 1 * (10 - 1) + 3 * 3 * (10 - 3) = 72$$

  to the total sum of distances.

And if for a column we have $|P_j(3)| = 2$ and $|P_j(2)| = 2$ then the column contributes

$$\sum_{m=0}^{3} m * |P_j(m)| * (|\mathcal{C}| - m) = 2 * 2 * (10 - 2) + 3 * 2 * (10 - 3) = 74$$

to the sum.

So each column can contribute maximal 74 and hence

$$N \leq 74 * 6 = 444.$$

So on the one hand we have $N \geq 450$ and on the other hand we have $N \leq 444$. This of course is not possible and therefore $\mathcal{C}$ is not a valid code. Thus we have

$$B_1(6, \mathcal{W}, 5) \leq 9.$$

□

Theorem 4.7. $B_1(6, \mathcal{W}, 5) = 9$ if $\{2, 3, 4\} \subseteq \mathcal{W}$.

Proof. To prove $B_1(6, \mathcal{W}, 5) = 9$ if $\{2, 3, 4\} \subseteq \mathcal{W}$ we will show that we can find an upper bound and lower bound with the same value, and so that that is the maximum value.

In Theorem 4.6 we found an upperbound and hence we know

$$B_1(6, \mathcal{W}, 5) \leq 9 \text{ if } \{2, 3, 4\} \subseteq \mathcal{W}.$$

For the lower bound we consider the following code:

$$\mathcal{C} = \{031320, 023012, 132132, 121203, 201031, 210302, 302313, 320121, 313230\}$$

this is a valid code in $\mathcal{B}_1(6, \mathcal{W})$ with $\{2, 3, 4\} \subseteq \mathcal{W}$ and with $d(\mathcal{C}) \geq 5$. We have $|\mathcal{C}| = 9$. Therefore we have

$$B_1(6, \mathcal{W}, 5) \geq 9 \text{ if } \{2, 3, 4\} \subseteq \mathcal{W}$$

.

Hence we can conclude

$$B_1(6, \mathcal{W}, 5) = 9 \text{ if } \{2, 3, 4\} \subseteq \mathcal{W}.$$

□

Theorem 4.8. $B_1(7, \mathcal{W}, 6) \leq 8$ for all $\mathcal{W}$.

Proof. To prove that $B_1(7, \mathcal{W}, 6) \leq 8$ we will use the same idea as the proof from Theorem 4.6.

Suppose we have a code $\mathcal{C} \subseteq B_1(7, \mathcal{W})$ with $d(\mathcal{C}) \geq 6$ and with $|\mathcal{C}| = 9$. Then because we have $|\mathcal{C}| = 9$ we need that for all $j \in \{1, \dots, 7\}$ we have $\sum_{k=0}^{3} |C_{j,k}| = 9$. Therefore we need that for all j we have

$$|P_j(3)| = 3 \text{ and } |P_j(0)| = 1 \text{ or}$$
$$|P_j(3)| = 2, |P_j(2)| = 1 \text{ and } |P_j(1)| = 1 \text{ or}$$
$$|P_j(3)| = 1 \text{ and } |P_j(2)| = 3.$$

Again we have this because for all j we have at most four different sets $C_{j,k}$ and for each such set we have $|C_{j,k}| \leq 3$.

Following the method of Plotkin [11] we will calculate the sum

$$N = \sum_{c_i \in \mathcal{C}} \sum_{c_l \in \mathcal{C}} d(c_i, c_l)$$

in two ways.

- Since $d(c_i, c_j) \geq d \; \forall c_i, c_l \in \mathcal{C}$ with $c_i \neq c_l$, we have

$$N \geq |\mathcal{C}|(|\mathcal{C}| - 1)d = 9 * 8 * 6 = 432.$$

- For the second way to calculate the sum N we look at the DNA-code in matrix form again and we look at how much each column can contribute to the total sum of distances.
  If for a column we have $|P_j(3)| = 3$ then this column contributes

$$\sum_{m=0}^{3} m * |P_j(m)| * (|\mathcal{C}| - m) = 3 * 3 * (9 - 3) = 54$$

  to the sum.
  If for a column we have $|P_j(3)| = 2$, $|P_j(2)| = 1$ and $|P_j(1)| = 1$ this column contributes

$$\sum_{m=0}^{3} m * |P_j(m)| * (|\mathcal{C}| - m) = 3 * 2 * (9 - 3) + 2 * 1 * (9 - 2) + 1 * 1 * (9 - 1) = 58$$

  to the sum.
  And if for a column we have $|P_j(3)| = 1$ and $|P_j(2)| = 3$ this column contributes

$$\sum_{m=0}^{3} m * |P_j(m)| * (|\mathcal{C}| - m) = 3 * 1 * (9 - 3) + 2 * 3 * (9 - 2) = 60$$

  to the sum.
  So each column can contribute maximal 60 and hence

$$N \leq 60 * 7 = 420.$$

So on the one hand we have $N \geq 432$ and on the other hand we have $N \leq 420$. This of course is not possible and therefore we $\mathcal{C}$ is not a valid code. Thus we have

$$B_1(7, \mathcal{W}, 6) \leq 8.$$

□

Theorem 4.9. $B_1(7, \mathcal{W}, 6) = 8$ if $\{3, 4\} \subseteq \mathcal{W}$.

Proof. To prove $B_1(7, \mathcal{W}, 6) = 8$ if $\{3, 4\} \subseteq \mathcal{W}$ we will show that we can find an upper bound and lower bound with the same value, and so that that is the maximum value.

In Theorem 4.8 we found an upper bound and hence we know

$$B_1(7, \mathcal{W}, 6) \leq 8 \text{ if } \{3, 4\} \subseteq \mathcal{W}.$$

For the lower bound we consider the following code:

$$\mathcal{C} = \{0210321, 0301230, 1312102, 1203013, 2030203, 2121312, 3132020, 3023131\}.$$

This is a valid code in $\mathcal{B}_1(7, \mathcal{W})$ with $\{3, 4\} \subseteq \mathcal{W}$ and with $d(\mathcal{C}) \geq 6$. We have $|\mathcal{C}| = 8$ and therefore

$$B_1(7, \mathcal{W}, 6) \geq 8 \text{ if } \{3, 4\} \subseteq \mathcal{W}.$$

Hence we can conclude

$$B_1(7, \mathcal{W}, 6) = 8 \text{ if } \{3, 4\} \subseteq \mathcal{W}.$$

□

Theorem 4.10. $B_1(8, \mathcal{W}, 7) \leq 5$ for all $\mathcal{W}$.

Proof. To prove that $B_1(8, \mathcal{W}, 7) \leq 5$ we will again use the same idea as the proof from Theorem 4.6

Suppose we have a code $\mathcal{C} \subseteq B_1(8, \mathcal{W})$ with $d(\mathcal{C}) \geq 7$ and with $|\mathcal{C}| = 6$. Then because we have $|\mathcal{C}| = 6$ we need that for all $j \in \{1, \dots, 8\}$ we have $\sum_{k=0}^{3} |C_{j,k}| = 6$. Therefore we need that for all j we have

$$|P_j(3)| = 2 \text{ and } |P_j(0)| = 2 \text{ or}$$

$$|P_j(3)| = 1 \text{ and } |P_j(1)| = 3 \text{ or}$$

$$|P_j(3)| = 1, |P_j(2)| = 1, |P_j(1)| = 1 \text{ and } |P_j(0)| = 1 \text{ or}$$

$$|P_j(2)| = 3 \text{ and } |P_j(0)| = 1 \text{ or}$$

$$|P_j(2)| = 2 \text{ and } |P_j(1)| = 2.$$

Again we have this because for all j we have at most four different sets $C_{j,k}$ and for each such set we have $|C_{j,k}| \leq 3$.

Following the method of Plotkin [11] we will calculate the sum

$$N = \sum_{c_i \in \mathcal{C}} \sum_{c_l \in \mathcal{C}} d(c_i, c_l)$$

in two ways.

- Since $d(c_i, c_j) \geq d \; \forall c_i, c_l \in \mathcal{C}$ with $c_i \neq c_l$, we have

$$N \geq |\mathcal{C}|(|\mathcal{C}| - 1)d = 6 * (6 - 1) * 7 = 210.$$

- For the second way to calculate the sum N we look at the DNA-code in matrix form again and we look at how much each column can contribute to the total sum of distances.

  If for a column we have $|P_j(3)| = 2$ then this column contributes

$$\sum_{m=0}^{3} m * |P_j(m)| * (|\mathcal{C}| - m) = 3 * 2 * (6 - 3) = 18$$

  to the sum.

  If for a column we have $|P_j(3)| = 1$ and $|P_j(1)| = 3$ then this column contributes

$$\sum_{m=0}^{3} m * |P_j(m)| * (|\mathcal{C}| - m) = 3 * 1 * (6 - 3) + 1 * 3 * (6 - 1) = 24$$

  to the sum.

  If for a column we have $|P_j(3)| = 1$, $|P_j(2)| = 1$ and $|P_j(1)| = 1$ then this column contributes

$$\sum_{m=0}^{3} m * |P_j(m)| * (|\mathcal{C}| - m) = 3 * 1 * (6 - 3) + 2 * 1 * (6 - 2) + 1 * 1 * (6 - 1) = 22$$

  to the sum.

If for a column we have $||P_j(2)| = 3$ then this column contributes

$$\sum_{m=0}^{3} m * |P_j(m)| * (|\mathcal{C}| - m) = 2 * 3 * (6 - 2) = 24$$

to the sum.

And if for a column we have $|P_j(2)| = 2$ and $|P_j(1)| = 2$ then this column contributes

$$\sum_{m=0}^{3} m * |P_j(m)| * (|\mathcal{C}| - m) = 2 * 2 * (6 - 2) + 1 * 2 * (6 - 1) = 26$$

to the sum.

So each column can contribute maximal 26 and hence

$$N \leq 26 * 8 = 208.$$

So on the one hand we have $N \geq 210$ and on the other hand we have $N \leq 208$. This of course is not possible and therefore $\mathcal{C}$ is not a valid code. Thus we have

$$B_1(8, \mathcal{W}, 7) \leq 5.$$

□

Theorem 4.11. $B_1(8, \mathcal{W}, 7) = 5$ if $\{3, 4, 5\} \subseteq \mathcal{W}$.

Proof. To prove $B_1(8, \mathcal{W}, 7) = 5$ if $\{3, 4, 5\} \subseteq \mathcal{W}$ we will show that we can find an upper bound and lower bound with the same value, and so that that is the maximum value.

In Theorem 4.10 we found an upper bound and hence we know

$$B_1(8, \mathcal{W}, 7) \leq 5 \text{ if } \{3, 4, 5\} \subseteq \mathcal{W}.$$

For the lower bound we consider the following code:

$$\mathcal{C} = \{01010232, 02101023, 03232101, 10123130, 20302312\}.$$

This is a valid code in $\mathcal{B}_1(8, \mathcal{W})$ with $\{3, 4, 5\} \subseteq \mathcal{W}$ and with $d(\mathcal{C}) \geq 7$ We have $|\mathcal{C}| = 5$ and therefore

$$B_1(8, \mathcal{W}, 7) \geq 5 \text{ if } \{3, 4, 5\} \subseteq \mathcal{W}.$$

Hence we can conclude

$$B_1(8, \mathcal{W}, 7) = 5 \text{ if } \{3, 4, 5\} \subseteq \mathcal{W}.$$

□

Theorem 4.12. For all $n, \mathcal{W}, r$ we have $B_r(n, \mathcal{W}, n - 1) \leq B_r(n - 1, \{0, \dots, n - 1\}, n - 2)$.

Proof. Suppose we have a DNA-code $\mathcal{C} \subseteq \mathcal{B}_r(n, \mathcal{W})$ with $d(\mathcal{C}) \geq n-1$ such that $|\mathcal{C}| > B_r(n-1, \{0, \dots, n-1\}, n - 2)$. If we remove the last symbol of every word in the code $\mathcal{C}$ then we have a new DNA-code $\mathcal{C}_1$ which has length $n - 1$, a run-length of r, all the words have weight $w \in \{0, \dots, n - 1\}$ and the new code has a minimum Hamming distance $d \geq n - 2$. But then we have $\mathcal{C}_1 \subseteq \mathcal{B}_r(n, \{0, \dots, n - 1\})$ with $d(\mathcal{C}_1) \geq n - 2$ and $|\mathcal{C}_1| = |\mathcal{C}| > B_r(n, \{0, \dots, n - 1\}, n - 2)$. This is a contradiction with the fact that $B_r(n, \{0, \dots, n-1\}, n-2)$ is the size of the largest DNA-code $\mathcal{B}_r(n, \{0, \dots, n-1\})$ with Hamming distance greater or equal then $n - 2$. And thus we need that

$$B_r(n, \mathcal{W}, n - 1) \leq B_r(n - 1, \{0, \dots, n - 1\}, n - 2).$$

□

Theorem 4.13. $B_1(9, \mathcal{W}, 8) = 5$ if $\{4, 5\} \subseteq \mathcal{W}$.

Proof. To prove $B_1(9, \mathcal{W}, 8) = 5$ if $\{4, 5\} \subseteq \mathcal{W}$ we will show that we can find an upper bound and a lower bound with the same value, and so that that value is the maximum value.

In Theorem 4.10 we found that $B_1(8, \mathcal{W}, 7) \leq 5 \; \forall \; \mathcal{W}$. This means $B_1(8, \{0, \dots, 8\}, 7) \leq 5$, therefore from Theorem 4.12 we know that we need $B_1(9, \mathcal{W}, 8) \leq 5$.

For the lower bound we consider the following code:

$$\mathcal{C} = \{023120230, 031212302, 120313123, 213032021, 302101313\}$$

this is a valid code in $\mathcal{B}_1(9, \mathcal{W})$ with $\{4, 5\} \subseteq \mathcal{W}$ and with $d(\mathcal{C}) \geq 8$. We have $|\mathcal{C}| = 5$ and therefore we have

$$B_1(9, \mathcal{W}, 8) \geq 5 \text{ if } \{4, 5\} \subseteq \mathcal{W}.$$

Hence we can conclude

$$B_1(9, \mathcal{W}, 8) = 5 \text{ if } \{4, 5\} \subseteq \mathcal{W}.$$

□

Theorem 4.14. $B_1(10, \mathcal{W}, 9) = 5$ if $\{4, 5, 6\} \subseteq \mathcal{W}$.

Proof. To prove $B_1(10, \mathcal{W}, 9) = 5$ if $\{4, 5, 6\} \subseteq \mathcal{W}$ we will show that we can find an upper bound and a lower bound with the same value, and so that that value is the maximum value.

In Theorem 4.10 we found that $B_1(8, \mathcal{W}, 7) \leq 5 \forall \mathcal{W}$, by Theorem 4.12 we found that $B_1(9, \mathcal{W}, 8) \leq 5$ and therefore by Theorem 4.12 it follows that we also have $B_1(10, \mathcal{W}, 9) \leq 5$.

For the lower bound we consider the following code:

$$\mathcal{C} = \{0231202303, 0312123021, 1203131230, 2130320212, 3021013132\},$$

this is a valid code in $\mathcal{B}_1(10, \mathcal{W})$ with $\{4, 5, 6\} \subseteq \mathcal{W}$ and with $d(\mathcal{C} \geq 9$. We have $|\mathcal{C}| = 5$ and therefore we have

$$B_1(10, \mathcal{W}, 9) \geq 5 \text{ if } \{4, 5, 6\} \subseteq \mathcal{W}.$$

Hence we can conclude

$$B_1(10, \mathcal{W}, 9) = 5 \text{ if } \{4, 5, 6\} \subseteq \mathcal{W}.$$

□

Theorem 4.15. $B_1(11, \mathcal{W}, 10) \leq 4$ for all $\mathcal{W}$.

Proof. To prove that $B_1(11, \mathcal{W}, 10) \leq 4$ we will again use the same idea as the proof from Theorem 4.6

Suppose we have a code $\mathcal{C} \subseteq B_1(11, \mathcal{W})$ with $d(\mathcal{C}) \geq 10)$ and with $|\mathcal{C}| = 5$. Then because we have $|\mathcal{C}| = 5$ we need that for all $j \in \{1, \dots, 11\}$ we have $\sum_{k=0}^{3} |C_{j,k}| = 5$. Therefore we need that for all j we have

$$|P_j(3)| = 1, \; |P_j(2)| = 1 \text{ and } |P_j(0)| = 2 \text{ or}$$
$$|P_j(3)| = 1, \; |P_j(1)| = 2 \text{ and } |P_j(0)| = 1 \text{ or}$$
$$|P_j(2)| = 2, \; |P_j(1)| = 1 \text{ and } |P_j(0)| = 1 \text{ or}$$
$$|P_j(2)| = 1 \text{ and } |P_j(1)| = 3.$$

Again we have this because for all j we have at most four different sets $C_{j,k}$ and for each such set we have $|C_{j,k}| \leq 3$.

Following the method of Plotkin [11] we will calculate the sum

$$N = \sum_{c_i \in \mathcal{C}} \sum_{c_l \in \mathcal{C}} d(c_i, c_l)$$

in two ways.

- Since $d(c_i, c_j) \geq d \; \forall c_i, c_l \in \mathcal{C}$ with $c_i \neq c_l$, we have

$$N \geq |\mathcal{C}|(|\mathcal{C}| - 1)d = 5 * (5 - 1) * 10 = 200.$$

- For the second way to calculate the sum N we look at the DNA-code in matrix form again and we look at how much each column can contribute to the total sum of distances.

  If for a column we have $P_j(3)| = 1$ and $|P_j(2)| = 1$ then this column contributes

$$\sum_{m=0}^{3} m * |P_j(m)| * (|\mathcal{C}| - m) = 3 * 1 * (5 - 3) + 2 * 1 * (5 - 2) = 12$$

  to the sum.

  If for a column we have $|P_j(3)| = 1$ and $|P_j(1)| = 2$ then this column contributes

$$\sum_{m=0}^{3} m * |P_j(m)| * (|\mathcal{C}| - m) = 3 * 1 * (5 - 3) + 1 * 2 * (5 - 1) = 14$$

  to the sum.

  If for a column we have $|P_j(2)| = 2$ and $|P_j(1)| = 1$ then this column contributes

$$\sum_{m=0}^{3} m * |P_j(m)| * (|\mathcal{C}| - m) = 2 * 2 * (5 - 2) + 1 * 1 * (5 - 1) = 16$$

  to the sum.

  And if for a column we have $|P_j(2)| = 1$ and $|P_j(1)| = 3$ then this column contributes

$$\sum_{m=0}^{3} m * |P_j(m)| * (|\mathcal{C}| - m) = 2 * 1 * (5 - 2) + 1 * 3 * (5 - 1) = 18$$

  to the sum.

  So each column can contribute maximal 18 and hence

$$N \leq 18 * 11 = 198.$$

So on the one hand we have $N \geq 200$ and on the other hand we have $N \leq 198$. This of course is not possible and therefore $\mathcal{C}$ is not a valid code. Thus we have

$$B_1(11, \mathcal{W}, 10) \leq 4.$$

□

Theorem 4.16. For $n \geq 11$ we have $B_1(n, \mathcal{W}, n - 1) \leq 4$ for all $\mathcal{W}$.

Proof. In Theorem 4.15 we saw that $B_1(n, \mathcal{W}, n - 1) \leq 4$ for $n = 11$. Now from Theorem 4.12 it follows that for $n \geq 11$ we have $B_1(n, \mathcal{W}, n - 1) \leq B_1(n - 1, \mathcal{W}, n - 2) \leq 4$. □

Theorem 4.17. If $n \geq 11$ and $\mathcal{W}$ such that for even-length codes we have $\frac{n}{2} \in \mathcal{W}$ and for odd-length codes we have $\frac{n-1}{2}, \frac{n+1}{2} \in \mathcal{W}$ then we have $B_1(n, \mathcal{W}, n - 1) = 4$.

Proof. In Theorem 4.16 we found that for $n \geq 11$ we have $B_1(n, \mathcal{W}, n-1) \leq 4 \; \forall \mathcal{W}$. From this we can conclude that if $n \geq 11$ and $\mathcal{W}$ is such that for even-length codes we have $\frac{n}{2} - 1, \frac{n}{2}, \frac{n}{2} + 1 \in \mathcal{W}$ and for odd-length codes we have $\frac{n-1}{2}, \frac{n+1}{2} \in \mathcal{W}$ then $B_1(n, \mathcal{W}, n-1) \leq 4$.
To find a lower bound, we will show that we can find a valid DNA-code consisting of 4 words. Consider the following code $\mathcal{C} = \{\mathbf{c}_1, \mathbf{c}_2, \mathbf{c}_3, \mathbf{c}_4\}$ in matrix form:

n

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

Figure 4.1: DNA-code $\mathcal{C}$ consisting of 4 words of length n.

Note that $\mathcal{C} \subseteq \mathcal{B}_1(n, \mathcal{W})$ with distance $d(\mathcal{C}) = n$ and with $\mathcal{W}$ as in the theorem. We thus found a valid DNA-code of size 4 where $d(\mathcal{C}) \geq n - 1$. So $B_1(n, \mathcal{W}, n-1) \geq 4$ Hence we can conclude that if $n \geq 11$ and $\mathcal{W}$ such that for even-length codes we have $\frac{n}{2} \in \mathcal{W}$ and for odd-length codes we have $\frac{n-1}{2}, \frac{n+1}{2} \in \mathcal{W}$ then we have

$$B_1(n, \mathcal{W}, n-1) = 4.$$

□

The following table will give an overview of the results we found in this chapter.

n	$B_1(n, \mathcal{W}, n-1)$
3	12
4	12
5	$10 \leq B_1(5, \mathcal{W}, 4) \leq 12$
6	9
7	8
8	5
9	5
10	5
≥ 11	4

Table 4.2: Overview of the sizes of the largest codes with words of length n where $d = n - 1$ and $\mathcal{W}$ is such that if n is even $\frac{n}{2} - 1, \frac{n}{2}, \frac{n}{2} + 1 \in \mathcal{W}$ and if n is odd $\frac{n-1}{2}, \frac{n+1}{2} \in \mathcal{W}$.

5

Algorithms to compute DNA-codes

In this thesis we are looking for the maximum sizes of DNA-codes. To find lower bounds for these values, we design algorithms that create valid DNA-codes and give the sizes of these codes. We present and analyze four algorithms from [1] and [3]. These algorithms have been modified to take into account an arbitrary run-length r and to take into account that the words must now satisfy one of the weights from the set $\mathcal{W}$. In addition, there are five other algorithms given, these are created by making small adjustments to the given algorithms in order to try to improve the lower bound. The python codes of these algorithms can be found in the Appendix A.

We will first give two definitions used in this section.

Definition 5.1. Suppose we have two DNA-words $\mathbf{x}$ and $\mathbf{y}$, then we say $\mathbf{y}$ is of higher **lexicographical** order if it holds that $x_j < y_j$ for $j = \min\{i \in \{1, \dots, n\} : x_i \neq y_i\}$.

Definition 5.2. Suppose we have two DNA words $\mathbf{x}$ and $\mathbf{y}$, then we say $\mathbf{y}$ is a $(d-1)$**-neighbour** of $\mathbf{x}$ if $d(\mathbf{x}, \mathbf{y}) \leq d - 1$.

5.1. Reference algorithms

5.1.1. Reference algorithm 1

In the first algorithm, an algorithm from [1], first a DNA-code $\mathcal{B}_r(n, \mathcal{W})$ is generated lexicographical. This code contains all the DNA words that have length n, every word has weight in $\mathcal{W}$ and has maximal run-length r. Also an empty list is created, at the end of the algorithm this list is the DNA-code. In every iteration the first word of the list with all the DNA-words is taken and is moved to the other list if the distance between this word and the words already in the list is minimal d. If the minimum distance breaks when moving, we delete the word from the list.

Algorithm 1: Reference algorithm 1

Data: Length n, GC-weight from $\mathcal{W}$, maximum run-length r and minimum Hamming distance d.
Result: A DNA-d code $\mathcal{C} \subseteq \mathcal{B}_r(n, \mathcal{W})$ with $d(\mathcal{C}) \geq d$.
1. Generate a list, *DNAcode*, with all the DNA words in lexicographical order from $\mathcal{B}_r(n, \mathcal{W})$.
2. Create an empty list *DNAdistance*.
if *DNAcode is not empty* **then**
  Define the first DNA word from *DNAcode* as **x**
  if **x** *satisfies* $d(\mathbf{x}, word) \geq d$ *for all* $word \in$ *DNAdistance* **then**
    x is appended to *DNAdistance*
  end
  x is deleted from *DNAcode*
else
  The DNA words in the list *DNAdistance* form the DNA-d code.
end

5.1.2. Reference algorithm 2

For the second algorithm, again an algorithm from [1], again the set with all the DNA words that have length n and where each DNA word has weight in $\mathcal{W}$ and has maximal run-length r is created. Then the algorithm checks for every word in this set how many other words are within $d-1$ distance of this word and creates two dictionaries with this information. In the first dictionary the key is the DNA-word and the value is all the $(d-1)$-neighbours. In the second dictionary the DNA words are the keys and its number of $(d-1)$-neighbours is the value. Also again an empty list is created. An iteration starts with adding the key with minimum value, so the word with the least $(d-1)$ neighbours, to the list. If there are multiple words with this same smallest value, then we choose the first one. Next we remove the word we added and its $(d-1)$-neighbours from the dictionary. In addition we also reduce the values of the keys that are $(d-1)$-neighbours of these words. The algorithm iterates until our dictionary is empty, then the list where we added the keys with minimum value to is now a valid DNA-code.

Algorithm 2: Reference algorithm 2

Data: Length n, GC-weight from $\mathcal{W}$, maximum run-length r and minimum Hamming distance d.
Result: A DNA-d code $\mathcal{C} \subseteq \mathcal{B}_r(n,\mathcal{W})$ with $d(\mathcal{C}) \geq d$.
1. Generate a list, *DNAcode*, with all the DNA words in lexicographical order from $\mathcal{B}_r(n,\mathcal{W})$.
2. Create an empty list *DNAdistance*.
3. Create a dictionary, *words_in_sphere*, where the keys are the DNA-words and the value is all its $(d-1)$-neighbours.
4. Create another dictionary, *distances*, where the key are the DNA-words and the value is its number of $(d-1)$-neighbours.

if *words_in_sphere is not empty* **then**
  $\mathbf{x}$ = first key with minimal value in *distances* ;
  Add $\mathbf{x}$ to *DNAdistance*;
  Delete $\mathbf{x}$ from *distances* and *words_in_sphere*;
  For all words with $d(\mathbf{x}, word) \leq d-1$ remove $word$ from *distances* and *words_in_sphere*;
  Update both the dictionaries *distances* and *words_in_sphere*;
else
  The words in the list *DNAdistance* form the DNA-d code.
end

5.1.3. Reference algorithm 3

Our third reference algorithm, an algorithm from [3], is very similar to the second algorithm. In this algorithm again at each iteration we find which word has the minimal $(d-1)$-neighbours, only instead of taking the first word with this minimal value, we now take the word in the middle of all the words with this minimal value. So at each iteration we find the minimal value in the dictionary where the keys are the DNA-words and the value is its number of $(d-1)$-neighbours. Next we find all keys from the dictionary, which value is this minimal value and we put these words in a list. Then we take the word which is in the middle of this list, and this is the word we add to our list.

Algorithm 3: Reference algorithm 3

Data: Length n, GC-weight from $\mathcal{W}$, maximum run-length r and minimum Hamming distance d.
Result: A DNA-d code $\mathcal{C} \subseteq \mathcal{B}_r(n, \mathcal{W})$ with $d(\mathcal{C}) \geq d$.

1. Generate a list, *DNAcode*, with all the DNA words in lexicographical order from $\mathcal{B}_r(n, \mathcal{W})$.
2. Create an empty list *DNAdistance*.
3. Create a dictionary, *words_in_sphere*, where the keys are the DNA-words and the value is all its $(d-1)$-neighbours.
4. Create another dictionary, *distances*, where the key are the DNA-words and the value is its number of $(d-1)$-neighbours.

```
if words_in_sphere is not empty then
    Create an empty list, minlst;
    Let minimum be the minimal value in distances;
    for key in distances do
        if key has minimum as its value then
            Append key to minlst
        end
    end
    Take the DNA word which is the middle value of the minlst let this be x;
    Add x to DNAdistance;
    Delete x from distances and words_in_sphere;
    For all words with d(x, word) ≤ d − 1 remove word from distances and words_in_sphere;
    Update both the dictionaries distances and words_in_sphere;
else
    The words in the list DNAdistance form the DNA-d code.
end
```

5.1.4. Reference algorithm 4

The fourth reference algorithm, again from [3], is very similar to Algorithm 3 from above. In this algorithm at each iteration, again a list with all DNA-words that have the minimum number of $(d-1)$-neighbours is created. In Algorithm 3 we took the middle word of this list and added it to our code. In algorithm 4 we will calculate for each of these words, how much the total distance between this word and all its $(d-1)$-neighbours is. Then we will take the word for which this number is maximal, and this is the word we will add to our code. To rephrase, from the list of words with minimal $(d-1)$-neighbours, we will take the word with maximal distance to its $(d-1)$-neighbours and we will add this word to our code.

Algorithm 4: Reference algorithm 4

Data: Length n, GC-weight from $\mathcal{W}$, maximum run-length r and minimum Hamming distance d.
Result: A DNA-d code $\mathcal{C} \subseteq \mathcal{B}_r(n, \mathcal{W})$ with $d(\mathcal{C}) \geq d$.

1. Generate a list, *DNAcode*, with all the DNA words in lexicographical order from $\mathcal{B}_r(n, \mathcal{W})$.
2. Create an empty list *DNAdistance*.
3. Create a dictionary, *words_in_sphere*, where the keys are the DNA-words and the value is all its $(d-1)$-neighbours.
4. Create another dictionary, *distances*, where the key are the DNA-words and the value is its number of $(d-1)$-neighbours.

if *words_in_sphere is not empty* **then**
  Create an empty list, *minlst*;
  Let *minimum* be the minimal value in *distances*;
  for *key in distances* **do**
    if *key has minimum as its value* **then**
      Append *key* to *minlst*
    end
  end
  for *word in minlst* **do**
    Add all the distances from word to its $(d-1)$neighbours
  end
  Get the word where this total disctance is maximal and let this word be $\mathbf{x}$;
  Add $\mathbf{x}$ to *DNAdistance*;
  Delete $\mathbf{x}$ from *distances* and *words_in_sphere*;
  For all words with $d(\mathbf{x}, word) \leq d-1$ remove $word$ from *distances* and *words_in_sphere*;
  Update both the dictionaries *distances* and *words_in_sphere*;
else
  The words in the list *DNAdistance* form the DNA-d code.
end

5.2. New algorithms

In Algorithm 1 and 2 we see that which DNA-word is added to our list to make a valid code, depends a lot on how the list from which we can take words is sorted. In the case of Algorithms 1 and 2 this is from a lexicographical sorted list. In case of Algorithm 1 we always take the first word and in Algorithm 2 we take the first word with the minimum number of $(d-1)$-neighbours.
In Algorithms 3 and 4 we see that we no longer take the first word, but we make a different choice. We make this choice of which word is going to be added to our code, in each iteration.

In this section we will try to improve the lower bounds by creating new algorithms. First we will create an algorithm that sorts the list with all the DNA-words before the iterations start. In addition, we present an algorithm that is very similar to Algorithms 3 and 4, but in which the word with minimal $(d-1)$-neighbours we choose, is chosen in again a different way.

5.2.1. Algorithms 5 and 6

In the algorithms we will present in this section we first sort the lexicographical list of all DNA words by weight. Such that a list is created with lists containing all the words with the same weight. Then we create an empty list, *sorted_DNAcode* , and from each list in the list with the words sorted by weight, we take one by one a DNA word and add this to the list *sorted_DNAcode*. This gives us a list with all the DNA words where the words alternately have a different weights.

Example 5.1. If for example we are trying to find a lower bound for $B_1(2, \{1, 2\}, 2)$. The lexicographic ordered list of all the words satisfying $r = 1$ and $w = 1$ or $w = 2$ would be:

$$\{02, 03, 12, 13, 20, 21, 23, 30, 31, 32\}$$

And the list ordered as described above would be:

$$\{02, 23, 03, 32, 12, 13, 20, 21, 30, 31\}$$

With this way of sorting, we modified Algorithms 1 and 2 and created Algorithms 5 and 6, respectively.

Algorithm 5:

Data: Length n, GC-weight from $\mathcal{W}$, maximum run-length r and minimum Hamming distance d.
Result: A DNA-d code $\mathcal{C} \subseteq \mathcal{B}_r(n, \mathcal{W})$ with $d(\mathcal{C}) \geq d$.
1. Generate a list, *DNAcode*, with all the DNA words in lexicographical order from $\mathcal{B}_r(n, \mathcal{W})$.
2. Sort the words of *DNAcode* per weight.
3. Create a list, *sorted_DNAcode* where the words are added to, one of each weight in turn.
4. Create an empty list *DNAdistance*.

if *sorted_DNAcode is not empty* **then**
  Define the first DNA word from *sorted_DNAcode* as **x**
  if $\mathbf{x}$ *satisfies* $d(\mathbf{x}, word) \geq d$ *for every* $word \in$ *DNAdistance* **then**
    $\mathbf{x}$ is appended to *DNAdistance*
  end
  $\mathbf{x}$ is deleted from *DNAcode*
else
  The DNA words in the list *DNAdistance* form the DNA-d code.
end

Algorithm 6:

Data: Length n, GC-weight from $\mathcal{W}$, maximum run-length r and minimum Hamming distance d.
Result: A DNA-d code $\mathcal{C} \subseteq \mathcal{B}_r(n, \mathcal{W})$ with $d(\mathcal{C}) \geq d$.
1. Generate a list, *DNAcode*, with all the DNA words in lexicographical order from $\mathcal{B}_r(n, \mathcal{W})$.
2. Sort the words of *DNAcode* per weight.
3. Create a list, *sorted_DNAcode* where the words are added to, one of each weight in turn.
4. Create an empty list *DNAdistance*.
5. Create a dictionary, *words_in_sphere*, where the keys are the DNA-words from *sorted_DNAcode* and the value is all its $(d-1)$-neighbours.
6. Create another dictionary, *distances*, where the key are the DNA-words and the value is its number of $(d-1)$-neighbours.

if *words_in_sphere is not empty* **then**
  $\mathbf{x} =$ first key with minimal value in *distances* ;
  Add $\mathbf{x}$ to *DNAdistance*;
  Delete $\mathbf{x}$ from *distances* and *words_in_sphere*;
  For all words with $d(\mathbf{x}, word) \leq d-1$ remove $word$ from *distances* and *words_in_sphere*;
  Update both the dictionaries *distances* and *words_in_sphere*;
else
  The words in the list *DNAdistance* form the DNA-d code.
end

5.2.2. Algorithms 7 and 8

In Algorithm 5 and 6 we looked at the algorithms if we sorted the list, with all the DNA words we can choose from, alternately by weight. Besides these algorithms we also created algorithms where the words are sorted in another way. In this section we will present these algorithms.

In these algorithm we sorted the words, such that the words with the same weight come one after the other. We sorted them in the way that you get a list where the words with the smallest weight come first and the weight of the words from the list get bigger and bigger. But we also sorted the other way around, where you first have the words of the greatest weight and then the words where the weight is less.

Example 5.2. In this example we consider again the case of Example 5.1, where we are interested in finding a lower bound for $B_1(2, \{1, 2\}, 2)$. When we now sort the list of all the words satisfying the

constraints in the way where the words with a lower weight come first we would get:

$$\{02, 03, 12, 13, 20, 21, 30, 31, 23, 32\}$$

And sorting in the way where the bigger weights come first, would give us:

$$\{23, 32, 02, 03, 12, 13, 20, 21, 30, 31\}$$

By sorting the words such that the words with lower weights come first and applying this to reference Algorithm 2, we created Algorithm 7. By sorting the words such that the words with greater weights come first, and applying this again to Algorithm 2, resulted in Algorithm 8.

Algorithm 7:

Data: Length n, GC-weight from $\mathcal{W}$, maximum run-length r and minimum Hamming distance d.
Result: A DNA-d code $\mathcal{C} \subseteq \mathcal{B}_r(n, \mathcal{W})$ with $d(\mathcal{C}) \geq d$.
1. Generate a list, *DNAcode*, with all the DNA words in lexicographical order from $\mathcal{B}_r(n, \mathcal{W})$.
2. Sort the words of *DNAcode* per weight.
3. Create a list, *sorted_DNAcode* where the words are added per weight, where the weights are getting bigger , beginning with the words with the lowest weight.
4. Create an empty list *DNAdistance*.
5. Create a dictionary, *words_in_sphere*, where the keys are the DNA-words from *sorted_DNAcode* and the value is all its $(d-1)$-neighbours.
6. Create another dictionary, *distances*, where the key are the DNA-words and the value is its number of $(d-1)$-neighbours.

if *words_in_sphere is not empty* **then**
  $\mathbf{x}$ = first key with minimal value in *distances* ;
  Add $\mathbf{x}$ to *DNAdistance*;
  Delete $\mathbf{x}$ from *distances* and *words_in_sphere*;
  For all words with $d(\mathbf{x}, word) \leq d-1$ remove $word$ from *distances* and *words_in_sphere*;
  Update both the dictionaries *distances* and *words_in_sphere*;
else
  The DNA words in the list *DNAdistance* form the DNA-d code.
end

Algorithm 8:

Data: Length n, GC-weight from $\mathcal{W}$, maximum run-length r and minimum Hamming distance d
Result: A DNA-d code $\mathcal{C} \subseteq \mathcal{B}_r(n, \mathcal{W})$ with $d(\mathcal{C}) \geq d$
1. Generate a list, *DNAcode*, with all the DNA words in lexicographical order from $\mathcal{B}_r(n, \mathcal{W})$
2. Sort the words of *DNAcode* per weight.
3. Create a list, *sorted_DNAcode* where the words are added per weight, where the weights are getting smaller, beginning with the words with the highest weight.
4. Create an empty list *DNAdistance*.
5. Create a dictionary, *words_in_sphere*, where the keys are the DNA-words from *sorted_DNAcode* and the value is all its $(d-1)$-neighbours.
6. Create another dictionary, *distances*, where the key are the DNA-words and the value is its number of $(d-1)$-neighbours.

if *words_in_sphere is not empty* **then**
  $\mathbf{x}$ = first key with minimal value in *distances* ;
  Add $\mathbf{x}$ to *DNAdistance*;
  Delete $\mathbf{x}$ from *distances* and *words_in_sphere*;
  For all words with $d(\mathbf{x}, word) \leq d-1$ remove $word$ from *distances* and *words_in_sphere*;
  Update both the dictionaries *distances* and *words_in_sphere*;
else
  The DNA words in the list *DNAdistance* form the DNA-d code.
end

5.2.3. Algorithm 9

Finally, in this section we present an algorithm very similar to Algorithms 2, 3 and 4. Just as in those algorithms, at each iteration we create a list with all the DNA words that have the minimal number of $(d-1)$- neighbours. In this algorithm, from this list we take the word that has minimal distance to the words that are already in this code. This is the word we add to our code.

Algorithm 9: Algorithm 9

Data: Length n, GC-weight from $\mathcal{W}$, maximum run-length r and minimum Hamming distance d
Result: A DNA-d code $\mathcal{C} \subseteq \mathcal{B}_r(n, \mathcal{W})$ with $d(\mathcal{C}) \geq d$
1. Generate a list, *DNAcode*, with all the DNA words in lexicographical order from $\mathcal{B}_r(n, \mathcal{W})$
2. Create an empty list *DNAdistance*.
3. Create a dictionary, *words_in_sphere*, where the keys are the DNA-words and the value is all its $(d-1)$-neighbours.
4. Create another dictionary, *distances*, where the key are the DNA-words and the value is its number of $(d-1)$-neighbours.

if *words_in_sphere is not empty* **then**
    Create an empty list, *minlst*;
    Let *minimum* be the minimal value in *distances*;
    for *key in distances* **do**
        if *key has minimum as its value* **then**
            Append *key* to *minlst*
        end
    end
    for *word in minlst* **do**
        Calculate the total distance between *word* and all the words in *DNAdistance*
    end
    Get the word where this total disctance is minimal and let this word be $\mathbf{x}$;
    Add $\mathbf{x}$ to*DNAdistance*;
    Delete $\mathbf{x}$ from *distances* and *words_in_sphere*;
    For all words with $d(\mathbf{x}, word) \leq d-1$ remove $word$ from *distances* and *words_in_sphere*;
    Update both the dictionaries *distances* and *words_in_sphere*;
else
    The words in the list *DNAdistance* form the DNA-d code
end

5.3. Evaluation of the algorithms

In this section we analyse the lower bounds we found for some cases with our different algorithms.

In the Tables 5.1, 5.2, 5.3 you see the results we found for the cases where $3 \leq n \leq 9$, $r = 1$,where the GC-weight is around half of the length of the code words and $d = 2$, $d = 3$ and $d = 4$ respectively. For the values of n, different lower bounds are obtained, the largest lower bound found with those algorithms is in bold.

What is noticeable in this tables is that the lower bounds found by the different algorithms are often quite far apart, especially when the values for d and n are further apart.

We also see that there is no algorithm that always find the largest lower bound. We see that for the cases in Table 5.1 and 5.2 the Algorithm 4 from Section 5.1.4 most often finds the largest lower bound. But in Table 5.3 we see that Algorithm 6 finds the largest lower bounds, except for the case where $n = 7$

In Table 5.4 we see the results for the case where we have n odd, $r = 1$, $d = 2$ and where $\mathcal{W}$ consists of only one of the weights, from the set of weights $\mathcal{W}$ we considered in Table 5.1. In this case, with only one of the weights, all the algorithms give the same lower bound. We notice that the lower bounds we found for the different values of n in Table 5.1 are not the same values of the cases in Table 5.4 for the same n with only one of the weights, added together. The lower bounds of Table 5.1 are not even that much larger then that of the ones with only one of the weights. This means that by considering codes where we accept words of different weights, we get a slightly larger code. But the largest code for a DNA code that contains words of two GC-weights is not simply all the code words of the

largest code of one of the weight together with all the code words of the largest code of the other weight.

In Table 5.5 we see the results for different values of n we found with the algorithms, for the cases where $r = 1$, the GC-weight is around half of the length of the code words and $d = n - 1$. This is the case we also considered in Chapter 4. We notice that the lower bounds for the cases where $n = 4$, $n = 5$, $n = 6$ and $n = 7$ we found in Chapter 4 are larger then the lower bounds the algorithms find for these cases.

Besides looking at the results for cases where we have $r = 1$, we also looked at the results of the algorithms when we considered codes where $r > 1$. The results we found are presented in Table 5.6, 5.8 and 5.7. As expected the values in Table 5.8 and 5.7 are the same. The algorithms for the same cases bit with $r = n - 1$ or $r = n$ find the same values. We also notice that this value is often the same for the different algorithms.

	Alg 1	Alg 2	Alg 3	Alg 4	Alg 5	Alg 6	Alg 7	Alg 8	Alg 9
$B_1(3,\{1,2\},2)$	9	11	11	**12**	11	**12**	**12**	**12**	11
$B_1(4,\{1,2,3\},2)$	34	35	**36**	**36**	32	35	**36**	**36**	33
$B_1(5,\{2,3\},2)$	76	**89**	86	85	79	86	88	88	85
$B_1(6,\{2,3,4\},2)$	274	303	316	**320**	242	300	**320**	318	292
$B_1(7,\{3,4\},2)$	576	664	657	**673**	594	667	651	668	658
$B_1(8,\{3,4,5\},2)$	2178	2261	2273	2316	2084	**2317**	2255	2281	2311
$B_1(9,\{4,5\},2)$	4576	5244	5224	**5383**	4775	5243	5141	5214	5283

Table 5.1: Lower bounds for $B_1(n,\mathcal{W},2)$ computed with the nine algorithms and where $\mathcal{W}$ is the set with the weights around half of the length of the code words. The values in bold are the largest lower bounds.

	Alg 1	Alg 2	Alg 3	Alg 4	Alg 5	Alg 6	Alg 7	Alg 8	Alg 9
$B_1(4,\{1,2,3\},3)$	9	**11**	**11**	10	10	**11**	**11**	**11**	**11**
$B_1(5,\{2,3\},3)$	22	26	26	26	22	26	25	25	**27**
$B_1(6,\{2,3,4\},3)$	54	63	**64**	**64**	55	62	63	62	60
$B_1(7,\{3,4\},3)$	134	**153**	152	150	134	**153**	149	**153**	151
$B_1(8,\{3,4,5\},3)$	351	399	401	**406**	352	403	404	398	405
$B_1(9,\{4,5\},3)$	862	976	971	**982**	878	965	977	975	972

Table 5.2: Lower bounds for $B_1(n,\mathcal{W},3)$ computed with the nine algorithms and where $\mathcal{W}$ is the set with the weights around half of the length of the code words. The values in bold are the largest lower bounds.

	Alg 1	Alg 2	Alg 3	Alg 4	Alg 5	Alg 6	Alg 7	Alg 8	Alg 9
$B_1(5,\{2,3\},4)$	8	**9**	**9**	**9**	7	**9**	**9**	**9**	**9**
$B_1(6,\{2,3,4\},4)$	18	**21**	20	**21**	17	**21**	**21**	**21**	**21**
$B_1(7,\{3,4\},4)$	45	**47**	46	**47**	42	44	46	44	46
$B_1(8,\{3,4,5\},4)$	104	119	116	117	105	**120**	**120**	119	116
$B_1(9,\{4,5\},4)$	242	271	271	274	248	**275**	272	272	274

Table 5.3: Lower bounds for $B_1(n,\mathcal{W},4)$ computed with the nine algorithms and where $\mathcal{W}$ is the set with the weights around half of the length of the code words. The values in bold are the largest lower bounds.

$B_1(3,\{1\},2)$	8
$B_1(3,\{2\},2)$	8
$B_1(5,\{2\},2)$	68
$B_1(5,\{3\},2)$	68
$B_1(7,\{3\},2)$	528
$B_1(7,\{4\},2)$	528
$B_1(9,\{4\},2)$	4336
$B_1(9,\{5\},2)$	4336

Table 5.4: Lower bounds for $B_1(n,\mathcal{W},2)$ where $\mathcal{W}$ consists of only one weight.

	Alg 1	Alg 2	Alg 3	Alg 4	Alg 5	Alg 6	Alg 7	Alg 8	Alg 9	Chap. 4
$B_1(3,\{1,2\},2)$	9	11	11	12	11	12	12	12	11	12
$B_1(4,\{1,2,3\},3)$	9	11	11	10	10	11	11	11	11	12
$B_1(5,\{2,3\},4)$	8	9	9	9	7	9	9	9	9	10
$B_1(6,\{2,3,4\},5)$	6	8	8	8	6	8	8	8	8	9
$B_1(7,\{3,4\},6)$	5	6	6	7	6	6	6	6	6	8
$B_1(8,\{3,4,5\},7)$	5	5	5	5	4	5	5	5	5	5
$B_1(9,\{3,4\},8)$	4	5	5	5	4	5	5	5	5	5

Table 5.5: Lower bounds for $B_1(n,\mathcal{W},n-1)$ where $\mathcal{W}$ is the set with the weights around half the length of the code words.

	Alg 1	Alg 2	Alg 3	Alg 4	Alg 5	Alg 6	Alg 7	Alg 8	Alg 9
$B_2(3,\{1,2\},2)$	**12**	**12**	**12**	**12**	**12**	**12**	**12**	**12**	**12**
$B_2(4,\{1,2,3\},3)$	54	**58**	54	**58**	51	57	**58**	55	54
$B_2(5,\{2,3\},4)$	**148**	144	139	141	137	142	141	143	144
$B_2(6,\{2,3,4\},5)$	768	783	774	784	692	782	752	752	**786**
$B_2(7,\{3,4\},6)$	**1864**	1661	1636	1670	1755	1623	1677	1667	1680

Table 5.6: Lower bounds for the cases where $r=2$. The values in bold are the largest lower bounds.

	Alg 1	Alg 2	Alg 3	Alg 4	Alg 5	Alg 6	Alg 7	Alg 8	Alg 9
$B_3(3,\{1,2\},2)$	**12**	**12**	**12**	**12**	**12**	**12**	**12**	**12**	**12**
$B_4(4,\{1,2,3\},3)$	**64**	**64**	**64**	**64**	57	**64**	**64**	**64**	**64**
$B_5(5,\{2,3\},4)$	**160**	**160**	145	146	156	**160**	**160**	**160**	**160**
$B_6(6,\{2,3,4\},5)$	**960**	**960**	**960**	**960**	914	**960**	**960**	**960**	**960**
$B_7(7,\{3,4\},6)$	**2240**	**2240**	2110	1915	2196	2190	**2240**	**2240**	**2240**
$B_8(8,\{3,4,5\},7)$	**14336**	**14336**	**14336**	**14336**	14100	**14336**	**14336**	**14336**	**14336**

Table 5.7: Lower bounds for the cases where $r=n$. The values in bold are the largest lower bounds.

	Alg 1	Alg 2	Alg 3	Alg 4	Alg 5	Alg 6	Alg 7	Alg 8	Alg 9
$B_2(3,\{1,2\},2)$	**12**	**12**	**12**	**12**	**12**	**12**	**12**	**12**	**12**
$B_3(4,\{1,2,3\},3)$	**64**	**64**	**64**	**64**	57	**64**	**64**	**64**	**64**
$B_4(5,\{2,3\},4)$	**160**	**160**	145	146	156	**160**	**160**	**160**	**160**
$B_5(6,\{2,3,4\},5)$	**960**	**960**	**960**	**960**	914	**960**	**960**	**960**	**960**
$B_6(7,\{3,4\},6)$	**2240**	**2240**	2110	1915	2196	2190	**2240**	**2240**	**2240**
$B_7(8,\{3,4,5\},7)$	**14336**	**14336**	**14336**	**14336**	14100	**14336**	**14336**	**14336**	**14336**

Table 5.8: Lower bounds for the cases where $r=n-1$. The values in bold are the largest lower bounds.

6

Conclusions and recommendations

In this chapter we will discuss the conclusions that can be deducted from the research done in this thesis. In addition also some recommendations for future research is discussed.

6.1. Conclusions

We defined $B_r(n, \mathcal{W}, d)$ in Chapter 2 as the size of the largest DNA-d code $\mathcal{C} \subseteq \mathcal{B}_r(n, \mathcal{W})$ with $d(\mathcal{C}) \geq d$. This code takes the parameters, length n of the code words, r the maximal run-length of the words, $\mathcal{W}$ as the set of GC-weights of the DNA code-words and d the minimum Hamming distance of the code. The aim of this thesis was to determine upper and lower bound of DNA codes with relaxed weight constraints. This relaxed weight constraint means that instead of all having the same GC-weight, code-words in a code can have a GC-weight from a set with multiple GC-weights.
Recall that from Chapter 3 we know that:

$$\forall r, n \geq 1, B_r(n, \mathcal{W}, n) = \begin{cases} 4 & \text{If there exists } w_1, w_2, w_3, w_4 \in \mathcal{W} \text{ such that } w_1 + w_2 + w_3 + w_4 = 2n \\ 2 & \text{If we have } \mathcal{W} \text{ such that there is no } w_1, w_2, w_3 \in \mathcal{W} \text{ with } n \leq w_1 + w_2 + w_3 \leq 2n \\ 3 & \text{otherwise} \end{cases} \tag{6.1}$$

In Chapter 4 we proved upper and lower bounds for DNA codes where we have a minimum Hamming distance of $d = n - 1$ and where the GC-weights of the words are around half the length n of the words. From this upper and lower bound we can make conclusions about the maximum size of these codes, these can be found in this table:

n	$B_1(n, \mathcal{W}, n-1)$
3	12
4	12
5	$10 \leq B_1(5, \mathcal{W}, 4) \leq 12$
6	9
7	8
8	5
9	5
10	5
≥ 11	4

Table 6.1: Overview of the sizes of the largest codes with words of length n where $d = n - 1$ and $\mathcal{W}$ is such that if n is even $\frac{n}{2} - 1, \frac{n}{2}, \frac{n}{2} + 1 \in \mathcal{W}$ and if n is odd $\frac{n-1}{2}, \frac{n+1}{2} \in \mathcal{W}$.

In addition to the lower bounds found in Chapters 3 and 4, we present and analyse algorithms in Chapter 5 that give valid DNA-d codes. The size of these codes are again lower bounds. From Tables 5.1, 5.2 and 5.3 we can conclude that we did not find an algorithm that always generates the largest possible code. From Tables 5.1 and 5.4 we can conclude that the largest code for a DNA code that contains words of two GC-weights is not simply all the code words of the largest code of one of the weight together with all the code words of the largest code of the other weight.

Table 6.2 gives the highest lower bounds for $B_1(n, \mathcal{W}, d)$, with $\mathcal{W}$ such that for even-length codes $\frac{n}{2}-1, \frac{n}{2}, \frac{n}{2}+1 \in \mathcal{W}$ and for odd-length codes $\frac{n-1}{2}, \frac{n+1}{2} \in \mathcal{W}$, found with the nine algorithms presented in Sections 5.1 and 5.2. We underline the values for which we proved in Chapter 4 that this is the largest possible size.

	$d=2$	$d=3$	$d=4$	$d=n-1$
$B_1(3,\{1,2\},d)$	<u>12</u>	-	-	<u>12</u>
$B_1(4,\{1,2,3\},d)$	36	11	-	11
$B_1(5,\{2,3\},d)$	89	27	9	9
$B_1(6,\{2,3,4\},d)$	320	64	21	8
$B_1(7,\{3,4\},d)$	673	153	47	7
$B_1(8,\{3,4,5\},d)$	2317	406	120	<u>5</u>
$B_1(9,\{4,5\},d)$	5383	982	275	<u>5</u>

Table 6.2: Highest lower bounds found with the presented algorithms

6.2. Recommendations

For future research there are multiple ideas.

In Chapter 4 we found for all $n \geq 3$ except for $n = 5$ the size of the largest possible code where $d = n - 1$ and where the weights of the DNA words are around half the length of the code words. It would be interesting to research the case where $n = 5$ more. It could be looked into if there is a way to improve the upper bound for this case, or if a valid DNA code can be found consisting of more than ten DNA words. As a result, the largest possible size could be found, or the range can be narrowed down.

Furthermore in this research we focused on DNA codes satisfying the no run-length constraint, the case where we have maximal run-length of $r = 1$. In future research it would be interesting to look at codes where we have relaxed weight constraints and where also the run-length constrained would be relaxed. In Section 5.3 we saw that the algorithms often find the same value for the different cases. This could indicate that this code size is the largest possible size. It would be interesting to research this further and to see if upper bounds can be found as well.

Also further improvements in the algorithms would be suggested. For the case with $d = n - 1$, analytical we found DNA codes of larger size than the algorithms did. This suggests that algorithms can be found that improve the existing lower bounds. For example, in Sections 5.2.1 and 5.2.2 two ways of sorting the lexicographical created list are considered, but other ways to sort this list could be considered. In addition, to allow computation of DNA codes with larger values of n the algorithms used could be optimised.

Bibliography

[1] C.J. van Leeuwen. *Constrained Codes for DNA-Based Storage Systems*. Bachelor Thesis, Delft University of Technology. May 2020.

[2] H. Vermeer. *Constrained Single-Error-Detecting codes for DNA-based Storage Systems*. Bachelor Thesis, Delft University of Technology. Feb. 2021.

[3] F.J. Laseur. *Constrained Error-Correcting Codes for DNA-Based Storage Systems*. Bachelor Thesis, Delft University of Technology. Dec. 2021.

[4] J. Bornholt et al. “A DNA-Based Archival Storage System”. In: *ACM SIGOPS Operating Systems Review* 50.2 (2016), pp. 637–649.

[5] S. M. H. T. Yazdi et al. “DNA-Based Storage: Trends and Methods”. In: *IEEE Transactions on Molecular, Biological and Multi-Scale Communications* 1.3 (2015), pp. 230–248.

[6] D. Limbachiya, M. K. Gupta, and V. Aggarwal. “Family of constrained codes for archival DNA data storage”. In: *IEEE Communications Letters* 22.10 (2018), pp. 1972–1975.

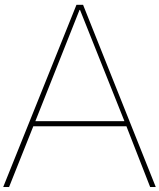
[7] D. Limbachiya, B. Rao, and M.K. Gupta. *The Art of DNA Strings: Sixteen Years of DNA Coding Theory*. July 1, 2016. URL: https://arxiv.org/abs/1607.00266.

[8] J. H. Weber, J. A. M. de Groot, and C. J. van Leeuwen. “On Single-Error-Detecting Codes for DNA-Based Data Storage”. In: *IEEE Communications Letters* 25.1 (2021), pp. 41–44.

[9] D.C. Hankerson and G. Hoffman. *Coding Theory and Cryptography*. 2nd ed. Abingdon, Verenigd Koninkrijk: Taylor Francis, 2000.

[10] O. D. King. “Bounds for DNA Codes with Constant GC-Content”. In: *The Electronic Journal of Combinatorics* 10.1 (2003).

[11] M. Plotkin. “Binary codes with specified minimum distance”. In: *IEEE Transactions on Information Theory* 6.4 (1960), pp. 445–450.



Python code

Reference algorithm 1

```
import itertools

def has_runlength(word):
    # checking the runlenght of a word
    runlength = 1
    for i in range(len(word)-1):
        run = 1
        while word[i] == word[i+1]:
            run += 1
            i +=1
            if i == (len(word)-1):
                break
        if run > runlength:
            runlength = run
    return runlength

def get_weight(word):
    # gets the GC-weight of a word
    return word.count(2) + word.count(3)

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

## Create a list with all the DNA-words
def DNA(n, lst_weight, r):
    Qcode = list(itertools.product(range(4), repeat=n))
    DNAcode = []
    for codeword in Qcode:
      if has_runlength(codeword) <= r and get_weight(codeword) in lst_weight:
            DNAcode.append(codeword)
    return DNAcode

# define the first word from the DNA list and add it to
# the list ”DNAcode” unless it breaks the minimal distance.
def step1(distancecode, DNAcode, d):
    word = DNAcode[0]
```

```
    if all(get_distance(word, codeword) >= d for codeword in distancecode):
        distancecode.append(word)
    DNAcode.remove(word)
    return distancecode, DNAcode

# keep calling step1 untill the list ”DNAcode” is empty.
def step2(distancecode, DNAcode, d):
    while DNAcode:
        step1(distancecode, DNAcode, d)
    return distancecode

# This function combines the functions above to create a DNA-d code.
def alg(n, lst_weight, r, d):
    DNAcode = DNA(n, lst_weight, r)
    DNAdistance = []
    step1a, step1b = step1(DNAdistance, DNAcode, d)
    step2_1 = step2(step1a, step1b, d)
    return step2_1

## This function will give the size of the DNA-d code we created
## and for each weight it will give
## the number of words with this weight in the code
def weights(n, lst_weight,r, d):
    lijst = alg(n, lst_weight,r, d)
    gewichten = []
    for word in lijst:
        gewichten.append(get_weight(word))

    weight_dict = {}
    weight_dict[’total’] = len(lijst)

    for w in lst_weight:
        weight_dict[w] = gewichten.count(w)

    return weight_dict
```

Reference algorithm 2

```
    import itertools

def has_runlength(word):
    # checking if a word has a runlength
    runlength = 1
    for i in range(len(word)-1):
        run = 1
        while word[i] == word[i+1]:
            run += 1
            i +=1
            if i == (len(word)-1):
                break
        if run > runlength:
            runlength = run
    return runlength

def get_weight(word):
    # gets the GC-weight of a word
    return word.count(2) + word.count(3)
```

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

def get_word_in_sphere(code, codeword, d):
    #gets all the (d-1)-neighbours (so the words to close)
    return [w for w in code if 0<get_distance(codeword, w)<d]

def get_minimum(distances):
    minimum = min(distances, key = distances.get)
    return minimum

# Creating our list with all the DNA-words
def DNA(n, lst_weight, r):
    Qcode = list(itertools.product(range(4), repeat=n))
    DNAcode = []
    for codeword in Qcode:
      if has_runlength(codeword) <= r and get_weight(codeword) in lst_weight:
            DNAcode.append(codeword)
    return DNAcode

# Creating dictionary with all the words and its (d-1)-neighbours
def words_list(code, d):
   words_in_sphere = {codeword: get_word_in_sphere(code, codeword, d) for codeword in code}
    return words_in_sphere

# Creating a dictionary with all the words and its number of (d-1)-neighbours
def words_dist(words_in_sphere):
    distances = {key: len(value) for key, value in words_in_sphere.items()}
    return distances

# Adding the word with minimal distance to the code
def code_list(distances, words_in_sphere, code):
    minimum = get_minimum(distances)
    code.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 code

# Keep calling code_list untill the ”words_in_spher” is empty
def calling(distances, words_in_sphere, code):
    while words_in_sphere:
        code_list(distances, words_in_sphere, code)
    return code

# This function combines the functions above to create a DNA-d code.
def algo(n, lst_weight,r, d):
    DNAcode = DNA(n, lst_weight,r)
```

```
    steplist = words_list(DNAcode, d)
    stepdist = words_dist(steplist)
    DNAdistance = []
    step3 = code_list(stepdist, steplist, DNAdistance)
    step4 = calling(stepdist, steplist, step3)
    return step4

## This function will give the size of the DNA-d code we created
## and for each weight it will give
## the number of words with this weight in the code
def weights(n, lst_weight,r, d):
    lijst = algo(n, lst_weight,r, d)
    gewichten = []
    for word in lijst:
        gewichten.append(get_weight(word))
    weight_dict = {}
    weight_dict['total'] = len(lijst)
    for w in lst_weight:
        weight_dict[w] = gewichten.count(w)
    return weight_dict
```

Reference algorithm 3

```
    import itertools

def has_runlength(word):
    # checking if a word has a runlength
    runlength = 1
    for i in range(len(word)-1):
        run = 1
        while word[i] == word[i+1]:
            run += 1
            i +=1
            if i == (len(word)-1):
                break
        if run > runlength:
            runlength = run
    return runlength

def get_weight(word):
    # gets the GC-weight of a word
    return word.count(2) + word.count(3)

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

def get_word_in_sphere(code, codeword, d):
    #gets all the (d-1)-neighbours (so the words to close)
    return [w for w in code if 0<get_distance(codeword, w)<d]

def get_minimum(distances):
    minimum = min(distances, key = distances.get)
    return minimum

# Creating our list with all the DNA-words
def DNA(n, lst_weight, r):
```

```
    Qcode = list(itertools.product(range(4), repeat=n))
    DNAcode = []
    for codeword in Qcode:
      if has_runlength(codeword) <= r and get_weight(codeword) in lst_weight:
            DNAcode.append(codeword)
    return DNAcode

# Creating dictionary with all the words and its (d-1)-neighbours
def words_list(code, d):
   words_in_sphere = {codeword: get_word_in_sphere(code, codeword, d) for codeword in code}
    return words_in_sphere

# Creating a dictionary with all the words and its number of (d-1)-neighbours
def words_dist(words_in_sphere):
    distances = {key: len(value) for key, value in words_in_sphere.items()}
    return distances

# Adding the word with minimal distance which is in the middle of the list
# with all the words with minimal distance to the code
def code_list(distances, words_in_sphere, code):
    minlst = []
    minimum = get_minimum(distances)
    for elt in distances:
        if distances[elt] == distances[minimum]:
            minlst.append(elt)
    minimum = minlst[int(len(minlst)/2)]
    code.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 code

# Keep calling code_list untill the ”words_in_spher” is empty
def calling(distances, words_in_sphere, code):
    while words_in_sphere:
        code_list(distances, words_in_sphere, code)
    return code

# This function combines the functions above to create a DNA-d code.
def algo(n, lst_weight,r, d):
    DNAcode = DNA(n, lst_weight,r)
    steplist = words_list(DNAcode, d)
    stepdist = words_dist(steplist)
    DNAdistance = []
    step3 = code_list(stepdist, steplist, DNAdistance)
    step4 = calling(stepdist, steplist, step3)
    return step4

## This function will give the size of the DNA-d code we created
## and for each weight it will give
```

```
## the number of words with this weight in the code
def weights(n, lst_weight,r, d):
    lijst = algo(n, lst_weight,r, d)
    gewichten = []
    for word in lijst:
        gewichten.append(get_weight(word))
    weight_dict = {}
    weight_dict['total'] = len(lijst)
    for w in lst_weight:
        weight_dict[w] = gewichten.count(w)
    return weight_dict
```

Refernce algorithm 4

```
    import itertools

def has_runlength(word):
    # returning the runlenght of a word
    runlength = 1
    for i in range(len(word)-1):
        run = 1
        while word[i] == word[i+1]:
            run += 1
            i +=1
            if i == (len(word)-1):
                break
        if run > runlength:
            runlength = run
    return runlength

def get_weight(word):
    # gets the GC-weight of a word
    return word.count(2) + word.count(3)

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

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

def get_maximum(distances):
    #returning the key with the max value of the dictionary distances
    maximum = max(distances, key=distances.get)
    return maximum

def get_minimum(distances):
    #returning the key with the max value of the dictionary distances
    minimum = min(distances, key = distances.get)
    return minimum

# Creating our list with all the DNA-words
def DNA(n, lst_weight, r):
    Qcode = list(itertools.product(range(4), repeat=n))
    DNAcode = []
    for codeword in Qcode:
```

```
      if has_runlength(codeword) <= r and get_weight(codeword) in lst_weight:
            DNAcode.append(codeword)
    return DNAcode

# Creating dictionary with all the words and its (d-1)-neighbours
def words_list(code, d):
   words_in_sphere = {codeword: get_word_in_sphere(code, codeword, d) for codeword in code}
    return words_in_sphere

# Creating a dictionary with all the words and its number of (d-1)-neighbours
def words_dist(words_in_sphere):
    distances = {key: len(value) for key, value in words_in_sphere.items()}
    return distances

# Adding the word with minimal distance which has maximal distance with its (d-
1)-neighbours
# with all the words with minimal distance to the code
def code_list(distances, words_in_sphere, code):
    minlst = []
    minimum = get_minimum(distances)
    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]
    code.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 code

# Keep calling code_list untill the ”words_in_spher” is empty
def calling(distances, words_in_sphere, code):
    while words_in_sphere:
        code_list(distances, words_in_sphere, code)
    return code

# This function combines the functions above to create a DNA-d code.
def algo(n, lst_weight,r, d):
    DNAcode = DNA(n, lst_weight,r)
```

```
    steplist = words_list(DNAcode, d)
    stepdist = words_dist(steplist)
    DNAdistance = []
    step3 = code_list(stepdist, steplist, DNAdistance)
    step4 = calling(stepdist, steplist, step3)
    return step4

## This function will give the size of the DNA-d code we created
## and for each weight it will give
## the number of words with this weight in the code
def weights(n, lst_weight,r, d):
    lijst = algo(n, lst_weight,r, d)
    gewichten = []
    for word in lijst:
        gewichten.append(get_weight(word))
    weight_dict = {}
    weight_dict['total'] = len(lijst)
    for w in lst_weight:
        weight_dict[w] = gewichten.count(w)

    return weight_dict
```

Algorithm 5

```
    import itertools


def has_runlength(word):
    # checking if a word has a runlength
    runlength = 1
    for i in range(len(word)-1):
        run = 1
        while word[i] == word[i+1]:
            run += 1
            i +=1
            if i == (len(word)-1):
                break
        if run > runlength:
            runlength = run
    return runlength

def get_weight(word):
    # gets the GC-weight of a word
    return word.count(2) + word.count(3)

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

def get_minimum(distances):
    minimum = min(distances, key = distances.get)
    return minimum

#Creating list with all the words that have length n,
#runlenght <=r and with a weight in lst_weight
def DNA(n, lst_weight, r):
    Qcode = list(itertools.product(range(4), repeat=n))
```

```
    DNAcode = []
    for codeword in Qcode:
      if has_runlength(codeword) <= r and get_weight(codeword) in lst_weight:
            DNAcode.append(codeword)
    return DNAcode

#Creating a list with per weight from lst_weight the words with this weight
def sorting_DNA(n, lst_weight, r):
    DNAcode = DNA(n, lst_weight,r)
    sorted_DNAcode = [[] for i in range(len(lst_weight))]
    for codeword in DNAcode:
        for i in range(len(lst_weight)):
            if get_weight(codeword) == lst_weight[i]:
                    sorted_DNAcode[i].append(codeword)
    return sorted_DNAcode

# Creating list where we in turns add words from a different weight
def sorting_list(n, lst_weight,r):
    code_sort = sorting_DNA(n, lst_weight,r)
    code= []
    lengte = 0
    for i in range(len(lst_weight)):
        if len(code_sort[i]) > lengte:
            lengte = len(code_sort[i])
    for j in range(lengte):
        for i in range(len(lst_weight)):
            if len(code_sort[i]) > 0:
                code.append(code_sort[i][0])
                del code_sort[i][0]
    return code

# define the first word from the DNA list and add it to
# the list ”DNAcode” unless it breaks the minimal distance.
def step1(distancecode, DNAcode, d):
    word = DNAcode[0]
    if all(get_distance(word, codeword) >= d for codeword in distancecode):
        distancecode.append(word)
    DNAcode.remove(word)
    return distancecode, DNAcode

# keep calling alg3_step1 untill the list ”DNAcode” is empty.
def step2(distancecode, DNAcode, d):
    while DNAcode:
        step1(distancecode, DNAcode, d)
    return distancecode

# This function combines the functions above to create a DNA-d code.
def alg3(n, lst_weight,r, d):
    DNAcode = sorting_list(n, lst_weight,r)
    DNAdistance = []
    step1a, step1b = step1(DNAdistance, DNAcode, d)
    step2_1 = step2(step1a, step1b, d)
    return step2_1

## This function will give the size of the DNA-d code we created
## and for each weight it will give
```

```
## the number of words with this weight in the code
def weights(n, lst_weight,r, d):
    lijst = alg3(n, lst_weight,r, d)
    gewichten = []
    for word in lijst:
        gewichten.append(get_weight(word))
    weight_dict = {}
    weight_dict['total'] = len(lijst)
    for w in lst_weight:
        weight_dict[w] = gewichten.count(w)
    return weight_dict
```

Algorithm 6

```
    import itertools

def has_runlength(word):
    # checking if a word has a runlength
    runlength = 1
    for i in range(len(word)-1):
        run = 1
        while word[i] == word[i+1]:
            run += 1
            i +=1
            if i == (len(word)-1):
                break
        if run > runlength:
            runlength = run
    return runlength

def get_weight(word):
    # gets the GC-weight of a word
    return word.count(2) + word.count(3)

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

def get_word_in_sphere(code, codeword, d):
    #gets all the (d-1)-neighbours (so the words to close)
    return [w for w in code if 0<get_distance(codeword, w)<d]

def get_maximum(distances):
#maximum is the key with the max value of the dictionary distances
    maximum = max(distances, key=distances.get)
    return maximum

def get_minimum(distances):
    minimum = min(distances, key = distances.get)
    return minimum

#Creating list with all the words that have length n,
#runlenght <=r and with a weight in lst_weight
def DNA(n, lst_weight, r):
    Qcode = list(itertools.product(range(4), repeat=n))
    DNAcode = []
    for codeword in Qcode:
```

```
      if has_runlength(codeword) <= r and get_weight(codeword) in lst_weight:
            DNAcode.append(codeword)
    return DNAcode

#Creating a list with per weight from lst_weight the words with this weight
def sorting_DNA(n, lst_weight, r):

    DNAcode = DNA(n, lst_weight,r)
    sorted_DNAcode = [[] for i in range(len(lst_weight))]
    for codeword in DNAcode:
        for i in range(len(lst_weight)):
            if get_weight(codeword) == lst_weight[i]:
                    sorted_DNAcode[i].append(codeword)
    return sorted_DNAcode

# Creating list where we in turns add words from a different weight
def sorting_list(n, lst_weight,r):
    code_sort = sorting_DNA(n, lst_weight,r)
    # code_sort.reverse()
    code= []
    lengte = 0
    for i in range(len(lst_weight)):
        if len(code_sort[i]) > lengte:
            lengte = len(code_sort[i])
    for j in range(lengte):
        for i in range(len(lst_weight)):
            if len(code_sort[i]) > 0:
                code.append(code_sort[i][0])
                del code_sort[i][0]
    return code

# Creating dictionary with all the words and its (d-1)-neighbours
def words_list(code, d):
   words_in_sphere = {codeword: get_word_in_sphere(code, codeword, d) for codeword in code}
    return words_in_sphere

# Creating a dictionary with all the words and its number of (d-1)-neighbours
def words_dist(words_in_sphere):
    distances = {key: len(value) for key, value in words_in_sphere.items()}
    return distances

# Adding the word with minimal distance to the code
def code_list(distances, words_in_sphere, code):
    minimum = get_minimum(distances)
    code.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 code
```

```
# Keep calling code_list untill the ”words_in_spher” is empty
def calling(distances, words_in_sphere, code):
    while words_in_sphere:
        code_list(distances, words_in_sphere, code)
    return code

# This function combines the functions above to create a DNA-d code.
def algo(n, lst_weight,r, d):
    DNAcode = sorting_list(n, lst_weight,r)
    steplist = words_list(DNAcode, d)
    stepdist = words_dist(steplist)
    DNAdistance = []
    step3 = code_list(stepdist, steplist, DNAdistance)
    step4 = calling(stepdist, steplist, step3)
    return step4

## This function will give the size of the DNA-d code we created
## and for each weight it will give
## the number of words with this weight in the code
def weights(n, lst_weight,r, d):
    lijst = algo(n, lst_weight,r, d)
    gewichten = []
    for word in lijst:
        gewichten.append(get_weight(word))
    weight_dict = {}
    weight_dict[’total’] = len(lijst)

    for w in lst_weight:
        weight_dict[w] = gewichten.count(w)
    return weight_dict
```

Algorithm 7

```
    import itertools

def has_runlength(word):
    # checking if a word has a runlength
    runlength = 1
    for i in range(len(word)-1):
        run = 1
        while word[i] == word[i+1]:
            run += 1
            i +=1
            if i == (len(word)-1):
                break
        if run > runlength:
            runlength = run
    return runlength

def get_weight(word):
    # gets the GC-weight of a word
    return word.count(2) + word.count(3)

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

```
def get_word_in_sphere(code, codeword, d):
    #gets all the (d-1)-neighbours (so the words to close)
    return [w for w in code if 0<get_distance(codeword, w)<d]

def get_maximum(distances):
    maximum = max(distances, key=distances.get)
    return maximum

def get_minimum(distances):
    minimum = min(distances, key = distances.get)
    return minimum

#Creating list with all the words that have length n,
# runlenght <=r and with a weight in lst_weight
def DNA(n, lst_weight, r):
    Qcode = list(itertools.product(range(4), repeat=n))
    DNAcode = []
    for codeword in Qcode:
      if has_runlength(codeword) <= r and get_weight(codeword) in lst_weight:
            DNAcode.append(codeword)
    return DNAcode

#Creating a list with per weight from lst_weight the words with this weight
def sorting_DNA(n, lst_weight, r):
    DNAcode = DNA(n, lst_weight,r)
    sorted_DNAcode = [[] for i in range(len(lst_weight))]
    for codeword in DNAcode:
        for i in range(len(lst_weight)):
            if get_weight(codeword) == lst_weight[i]:
                    sorted_DNAcode[i].append(codeword)
    return sorted_DNAcode

# Creating a list where we the words with same weight come after each other
def sorting_list(n, lst_weight,r):
    code_sort = sorting_DNA(n,lst_weight,r)
    code = []
    for i in range(len(lst_weight)):
        for j in range(len(code_sort[i])):
            code.append(code_sort[i][j])
    return code

# Creating dictionary with all the words and its (d-1)-neighbours
def words_list(code, d):
   words_in_sphere = {codeword: get_word_in_sphere(code, codeword, d) for codeword in code}
    return words_in_sphere

# Creating a dictionary with all the words and its number of (d-1)-neighbours
def words_dist(words_in_sphere):
    distances = {key: len(value) for key, value in words_in_sphere.items()}
    return distances

# Adding the word with minimal distance to the code
def code_list(distances, words_in_sphere, code):
    minimum = get_minimum(distances)
    code.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 code

# Keep calling code_list untill the ”words_in_spher” is empty
def calling(distances, words_in_sphere, code):
    while words_in_sphere:
        code_list(distances, words_in_sphere, code)
    return code

# This function combines the functions above to create a DNA-d code.
def algo(n, lst_weight,r, d):
    DNAcode = sorting_list(n, lst_weight,r)
    steplist = words_list(DNAcode, d)
    stepdist = words_dist(steplist)
    DNAdistance = []
    step3 = code_list(stepdist, steplist, DNAdistance)
    step4 = calling(stepdist, steplist, step3)
    return step4

## This function will give the size of the DNA-d code we created
## and for each weight it will give
## the number of words with this weight in the code
def weights(n, lst_weight,r, d):
    lijst = algo(n, lst_weight,r, d)
    gewichten = []
    for word in lijst:
        gewichten.append(get_weight(word))
    weight_dict = {}
    weight_dict[’total’] = len(lijst)

    for w in lst_weight:
        weight_dict[w] = gewichten.count(w)
    return weight_dict
```

Algorithm 8

```
    import itertools

def has_runlength(word):
    # checking if a word has a runlength
    runlength = 1
    for i in range(len(word)-1):
        run = 1
        while word[i] == word[i+1]:
            run += 1
            i +=1
            if i == (len(word)-1):
                break
        if run > runlength:
            runlength = run
```

```
    return runlength

def get_weight(word):
    # gets the GC-weight of a word
    return word.count(2) + word.count(3)

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

def get_word_in_sphere(code, codeword, d):
    #gets all the (d-1)-neighbours (so the words to close)
    return [w for w in code if 0<get_distance(codeword, w)<d]

def get_maximum(distances):
    maximum = max(distances, key=distances.get)
    return maximum

def get_minimum(distances):
    minimum = min(distances, key = distances.get)
    return minimum

#Creating list with all the words that have length n,
#runlenght <=r and with a weight in lst_weight
def DNA(n, lst_weight, r):
    Qcode = list(itertools.product(range(4), repeat=n))
    DNAcode = []
    for codeword in Qcode:
      if has_runlength(codeword) <= r and get_weight(codeword) in lst_weight:
            DNAcode.append(codeword)
    return DNAcode

#Creating a list with per weight from lst_weight the words with this weight
def sorting_DNA(n, lst_weight, r):
    DNAcode = DNA(n, lst_weight,r)
    sorted_DNAcode = [[] for i in range(len(lst_weight))]
    for codeword in DNAcode:
        for i in range(len(lst_weight)):
            if get_weight(codeword) == lst_weight[i]:
                    sorted_DNAcode[i].append(codeword)
    return sorted_DNAcode

# Creating a list where we the words with same weight come after each other
def sorting_list(n, lst_weight,r):
    code_sort = sorting_DNA(n,lst_weight,r)
    code = []
    for i in range(len(lst_weight)):
        for j in range(len(code_sort[-(i+1)])):
            code.append(code_sort[-(i+1)][j])
    return code

# Creating dictionary with all the words and its (d-1)-neighbours
def words_list(code, d):
   words_in_sphere = {codeword: get_word_in_sphere(code, codeword, d) for codeword in code}
    return words_in_sphere
```

```
# Creating a dictionary with all the words and its number of (d-1)-neighbours
def words_dist(words_in_sphere):
    distances = {key: len(value) for key, value in words_in_sphere.items()}
    return distances

# Adding the word with minimal distance to the code
def code_list(distances, words_in_sphere, code):
    minimum = get_minimum(distances)
    code.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 code

# Keep calling code_list untill the ”words_in_spher” is empty
def calling(distances, words_in_sphere, code):
    while words_in_sphere:
        code_list(distances, words_in_sphere, code)
    return code

# This function combines the functions above to create a DNA-d code.
def algo(n, lst_weight,r,d):
    DNAcode = sorting_list(n, lst_weight,r)
    steplist = words_list(DNAcode, d)
    stepdist = words_dist(steplist)
    DNAdistance = []
    step3 = code_list(stepdist, steplist, DNAdistance)
    step4 = calling(stepdist, steplist, step3)
    return step4

## This function will give the size of the DNA-d code we created
## and for each weight it will give
## the number of words with this weight in the code
def weights(n, lst_weight,r, d):
    lijst = algo(n, lst_weight,r,d)
    gewichten = []
    for word in lijst:
        gewichten.append(get_weight(word))
    weight_dict = {}
    weight_dict[’total’] = len(lijst)
    for w in lst_weight:
        weight_dict[w] = gewichten.count(w)
    return weight_dict
```

Algorithm 9

```
    import itertools

def has_runlength(word):
    # returning the runlenght of a word
    runlength = 1
```

```
    for i in range(len(word)-1):
        run = 1
        while word[i] == word[i+1]:
            run += 1
            i +=1
            if i == (len(word)-1):
                break
        if run > runlength:
            runlength = run
    return runlength

def get_weight(word):
    # gets the GC-weight of a word
    return word.count(2) + word.count(3)

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

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

def get_maximum(distances):
    #returning the key with the max value of the dictionary distances
    maximum = max(distances, key=distances.get)
    return maximum

def get_minimum(distances):
    #returning the key with the max value of the dictionary distances
    minimum = min(distances, key = distances.get)
    return minimum

# Creating our list with all the DNA-words
def DNA(n, lst_weight, r):
        Qcode = list(itertools.product(range(4), repeat=n))
    DNAcode = []
    for codeword in Qcode:
      if has_runlength(codeword) <= r and get_weight(codeword) in lst_weight:
            DNAcode.append(codeword)
    return DNAcode

# Creating dictionary with all the words and its (d-1)-neighbours
def words_list(code, d):
   words_in_sphere = {codeword: get_word_in_sphere(code, codeword, d) for codeword in code}
    return words_in_sphere

# Creating a dictionary with all the words and its number of (d-1)-neighbours
def words_dist(words_in_sphere):
    distances = {key: len(value) for key, value in words_in_sphere.items()}
    return distances

# Adding the word with minimal distance which has minimal distance
# to all the words which are already in the code
def code_list(distances, words_in_sphere, code):
    minlst = []
```

```
    minimum = get_minimum(distances)
    for elt in distances:
        if distances[elt] == distances[minimum]:
            minlst.append(elt)
    distances_from_code = []
    for i in range(len(minlst)):
        dist = 0
        for element in code:
            dist += get_distance(minlst[i], element)
        distances_from_code.append(dist)

    mn = min(distances_from_code)
    index = distances_from_code.index(mn)
    minimum = minlst[index]
    code.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 code

# Keep calling code_list untill the ”words_in_sphere” is empty
def calling(distances, words_in_sphere, code):
    while words_in_sphere:
        code_list(distances, words_in_sphere, code)
    return code

# This function combines the functions above to create a DNA-d code.
def algo(n, lst_weight,r, d):
    DNAcode = DNA(n, lst_weight,r)
    steplist = words_list(DNAcode, d)
    stepdist = words_dist(steplist)
    DNAdistance = []
    step3 = code_list(stepdist, steplist, DNAdistance)
    step4 = calling(stepdist, steplist, step3)
    return step4

## This function will give the size of the DNA-d code we created
## and for each weight it will give
## the number of words with this weight in the code
def weights(n, lst_weight,r, d):
    lijst = algo(n, lst_weight,r, d)
    gewichten = []
    for word in lijst:
        gewichten.append(get_weight(word))
    weight_dict = {}
    weight_dict[’total’] = len(lijst)
    for w in lst_weight:
        weight_dict[w] = gewichten.count(w)
    return weight_dict
```