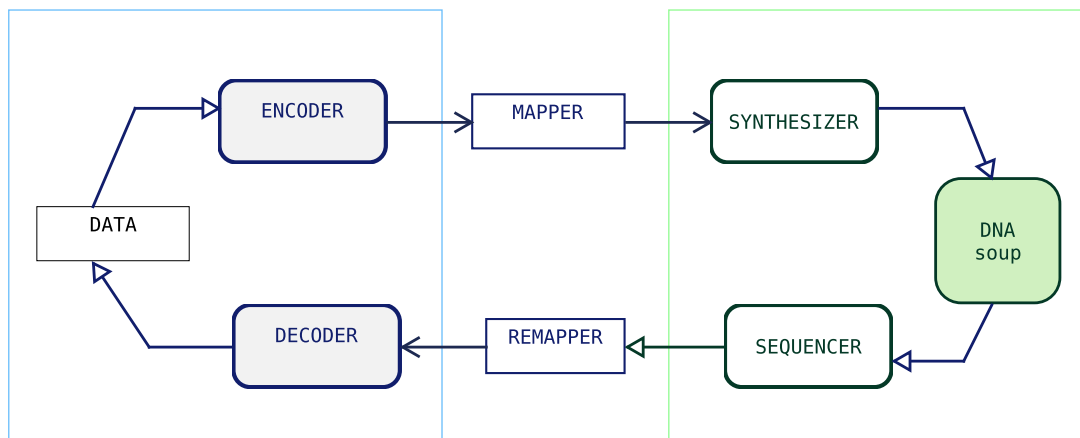

... 2.”

Master Thesis

DNA as a storage - Error Detection based on Cyclic Codes

Fikrat Talibli

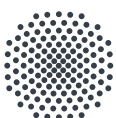


Date of hand out: November 16, 2019

Date of hand in: July 13, 2020

Supervisor: Dr -Ing. Christian
Senger

submitted to



University of Stuttgart
Institute of Telecommunications

Abstract

Due to the constant information generation, data centres are taking more and more space. State of the art technology allows storing data on multiple servers using cloud computing management. Even though the name says cloud, there are buildings of hardware behind the term. However, early on, scientists proposed yet another marvellous solution, which is to use DNA for storing data. In 2012 when the creation of synthetic molecules was possible using just a sequence of DNA, it became clear that one can utilize technology for data storage. Reading DNA from the sample was already possible from the 2000s. Combined with channel coding, this puzzle might solve many problems for data storage. DNA molecules have such properties as capacity, where 3 grams of DNA can store up to 6.4 Zettabytes and stability and DNA can be recovered after many years.

The motivation for selecting this topic was based upon the properties. DNA offers as a storage medium and exciting innovation that is both risky and groundbreaking. This work compiles state-of-the-art schematics behind DNA as storage and offers a particular technique for sequencing. Additionally, codes contain simulation for the proposed method and tests on a randomized dataset.

Title page image: Basic schema of DNA as a storage. Schema is component based, where blue part corresponds to Error Control Codes(storage) and green corresponds to Biology(storage medium).

Acknowledgements

First of all, I would like to express my gratitude to Dr. -Ing. Christian Senger for giving me the chance to work on a master thesis with this exciting topic. I am grateful for your guidance, patience and dedication throughout the thesis. I would also like to thank Prof. Dr. rer. nat. Björn Voß for listening to the ideas on the topic as a biologist.

Additionally, I would like to thank my parents(Arzu and Talib Talibli) and my brother(Farid Talibli) for their support.

Contents

Notations	VI
1. Introduction	1
1.1. Motivation	1
1.2. Aim of this work	2
1.3. Chapters review	2
2. Biology for Electrical Engineers	4
2.1. Properties of DNA	4
2.2. Introduction to DNA	4
2.3. Synthesis	5
2.4. Sequencing	6
2.5. Putting biological part altogether	8
3. Error Control Codes	9
3.1. Introduction	9
3.2. Forward error correction	9
3.3. Finite Fields and Mapping	10
3.4. Error Detection	10
3.5. Putting error control codes altogether	12
4. Workflow of DNA as a storage	14
4.1. Schema of DNA as a storage	14
4.2. Tools used for DNA as a storage	16
4.2.1. Error control codes package	17
4.2.2. Bioinformatics package	17
4.2.3. Helpers package	19
4.2.4. Workflow manager	19
4.3. Sequencing as a computational problem	20
4.3.1. Exploding newspaper	21
4.3.2. Shortest common superstring and de Bruijn graph	22
4.4. Error detection for sequenced reads	25
5. Workflow of DNA as a storage	29
5.1. Reads vs. Codewords	29
5.2. Concept of headers and proto messages	30

5.3. Assembling DNA sequence using valid codewords	32
5.4. Empirical analysis on toy example	36
5.4.1. Updated setup	36
5.4.2. Simulations performed on toy parameterized example	38
5.5. Conclusion	41
6. Future outlook	43
A. Appendix	44
A.1. Tools used for DNA as a storage	44
A.1.1. Error control codes package	44
A.1.2. Error control codes package	46
Bibliography	47

Notations

x	scalar variables, especially in time domain: italic lower case letters
X	scalar variables, especially in frequency domain or random variables: italic upper case letters
\mathbf{x}	vector variables: boldface upright lower case letters
\mathbf{X}	matrix variables: boldface upright upper case letters
$x(t), H(z)$	functions of continuous variables: argument is placed in round parentheses
$x[k]$	functions of discrete variables: argument is placed in brackets
x_k	scalar elements of a vector or element of a sequence: element index is a subscript
$X_{r,c}$	scalar elements of a matrix: row and column indices are subscripts
$a \cdot b, \mathbf{a} \cdot \mathbf{b}, \mathbf{H} \cdot \mathbf{x}$	scalar product of two scalars, vectors or matrices
$\mathbf{a} \times \mathbf{b}, 5.1 \times 10^{-9}$	cross product of two vectors or scientific notation of numbers
$a * b$	linear convolution
$a \otimes b$	circular convolution
$P[X < 0]$	Probability of an event on a random variable X
$P[X = x Y = y], P[x y]$	Probability of the event $X = x$ under the condition $Y = y$
$p_X(x), p(x)$	continuous probability density function of a random variable X
$p_{X Y=y}(x), p(x y)$	probability density function of a random variable X under the condition $Y = y$

$E[X]$	expected value of a random variable X
$X(\omega) = \mathcal{F}\{x(t)\},$ $X(\omega) \overset{\circ}{\longrightarrow} \bullet x(t)$	Fourier transform from time-domain signal $x(t)$ into frequency-domain spectrum $X(\omega)$

1. Introduction

This chapter serves gives brief description of state-of-the-art research in the field of DNA as a storage. Motivation, aims of this work and thesis structure are described in this chapter.

1.1. Motivation

In today's world of constant information generation, more and more data centres are built. Moreover, it is predicted that by 2025 global datasphere will hit 175 billion terabytes [1]. Even though there is tremendous development in cloud technology(distributed systems or modern software-defined networking), they are notably capacious. There are some aspects of risk involved in keeping data centers secure and safe: environmental changes, animals, insects and security breaches.

One of the alternate storage mediums that scientists considered is synthesized DNA molecules [2]. Recent research showed experimentally that it is indeed possible to store data in DNA. As an example, research teams from the University of Illinois were able to store the files in a jar with synthetic DNA molecules [3]. Additionally, researchers at Microsoft and the University of Washington have demonstrated the first fully automated read-only DNA system in artificially created - synthetic DNA [4]. Following is a direct quote from Microsoft team: This is a key step towards moving new technology from research laboratories to commercial data centres. The main reason why scientists decided to look at DNA as an alternate storage medium is due to the properties it possesses. Unique properties of DNA(in terms of storage) are:

- DNA molecules can potentially store the entire volume of global digital information in about 9 litres of DNA solution for millennia as it is described in the article from technology magazine Habr. The capacity of DNA where one cell with certain DNA and mass of 3 pgrams stores 6.4Gbs of data [3](which makes 1 gram equivalent to 2.1 Zettabytes);
- Stability is another property that DNA possesses [5]. It is a great medium for long-term storage. Research conducted in ETH Zurich states that if synthetic DNA molecules are saved in a specific material and at a certain low temperature, then the information will be stored for millions of years [5].
- Finally, self-assembly potential - DNA is proven as the building blocks of small self-assembly based chips [6].

Interestingly enough in human body DNA in a certain way is used as a hereditary data storage. In nature, DNA molecules carry genetic instructions that control the functioning of living organisms [7].

1.2. Aim of this work

Despite research in both theoretical and experimental worlds, there is no clear step-by-step guide of how exactly DNA storage works, what components are necessary to create even simulation of the working system. Additionally, there is no comprehensive description of biological components and sequencing algorithms for reading DNA back and assembling it for electrical engineers. The thesis provides compiled comprehensive guide for electrical engineers to jump-start the topic of DNA as storage and basic prerequisite biology and computer science. Additionally, the thesis provides a virtual simulation of the system on python.

The popularity of modern second-generation sequencers(sequencer reads DNA from the unknown sample of molecules) increases and almost every sequencing lab is using them. Even though the accuracy of DNA sequencers to read back genome without errors increases(98%), they are still imposing problems when sequencing long strands of DNA sample by using reads longer than 300 characters. This work also introduces a unique way of encoding data, such that when collected back together, it is possible to detect erroneous parts of sequenced DNA. The empirical analysis was conducted on toy-parameter example, where parameters were assumed to be proportional to real-world parameters.

A thesis serves as a comprehensive guide for communication specialists or bioinformaticians on basics of DNA as a storage.

1.3. Chapters review

This work is divided into six parts. Chapter one provides a quick introduction to DNA as a storage - motivation (1.1) behind considering DNA as a storage medium, goal of this work (1.2) and project structure (1.3).

Chapter two provides comprehensive introduction to biology, that is required for this work. Chapter two describes unique properties (2.1) of DNA, introduction to DNA (2.2) for electrical engineers, biological synthesis (2.3) and sequencing (2.4). Chapter is finalized by putting biological/biochemical interface altogether (2.5).

Chapter three provides all necessary background information in error control codes required for assembling model of DNA as a storage system. Chapter starts with brief introduction to coding theory (3.1), follows by brief description types error control code types (3.2), finite

fields and mapping technique (3.3) used for (ECC \leftrightarrow DNA), brief description of error detection techniques and CRC codes (3.4). Chapter is finalized by putting error control codes altogether (3.5).

Chapter four introduces schematics of full blown DNA as a storage system. Chapter introduces the schematics of DNA as a storage (4.1); follows by description of tools used for project (4.2), concrete schemas of each package implemented for bioinformatics (4.2.1), ECC respectively (4.2.2) and helper interfaces (4.2.3), and workflow manager class (4.2.4). Chapter proceeds with description of sequencing as a computational problem (4.3), where sequencing is first associated with exploding newspaper problem (4.3.1) and then exploited using graph algorithms, such as shortest common superstring and de Bruijn graph (4.3.2). Chapter is finalized by introducing ECC detection mechanism for erroneous reads - this section also serves as opening for the next chapter(4.4).

Chapter five continues the topic of error detection techniques for sequencing. The chapter introduces the differences and similarities between codewords and reads (4.1); follows by the introduction of headers and proto messages (4.2.1), where codewords are adapted to reads format for further assembly. The chapter proceeds with an assembly of the genome using valid codewords (4.3), where reads are represented as codewords with a header for providing overlap. The chapter is finalized by providing an empirical analysis of the system(simulated mini-system/"toy" system) that compares the usage of proto messages and original messages (4.4) and provides comprehensive "assembly accuracy" graph.

2. Biology for Electrical Engineers

Before going into the schematics and details of DNA as storage, it is necessary to understand the motivation behind using DNA and basic terminology from classic and synthetic biology. This chapter will describe significant reasons behind choosing DNA as a storage medium and will serve as a sufficient head start in biology for electrical engineers.

2.1. Properties of DNA

In today's world of constant information generation, data centers are built. Moreover, at the rate that data gets generated, it is predicted that by 2025 global datasphere will hit 175 billion terabytes. Additionally, problems with state-of-the-art data storage technologies is a lifetime of solid-state drives and hard disks. According to several articles it is limited to approximately ten years. Researchers decided to consider DNA as one of the possible storage mediums.

The work discusses DNA as a storage medium due to the unique properties of DNA. Unique properties of DNA are:

- capacity, where one cell with certain DNA and mass of 3 pgrams stores 6.4Gbs of data(which makes 1 gram equivalent to 2.1 Zettabytes);
- stability - it was possible to read out the DNA of mammoth;
- self-assembly potential - DNA is proven as building block of small self-assembly based chips.

Interestingly in the human body, DNA in a certain way is used as a data storage. It carries genetic information - instructions to carry out vital biological processes.

2.2. Introduction to DNA

DNA(deoxyribonucleic acid) - is a *macromolecule(biopolymer)* that consists of smaller molecules (monomers) - nucleotides [8]. Nucleotides consist of three main parts: nitrogenous base, sugar base and phosphate group. Electrical engineers and channel coding specialists are interested only in nitrogenous bases. Nitrogenous base is a molecule that contains nitrogen and chemical base; nitrogenous bases are building blocks of DNA and for electrical engineers they are simply considered to be characters/symbols from alphabet of {**A,C,G,T**} - {Adenine, Cytosine,

Guanine, Thymine}.

DNA sequence or strand - is *one side of DNA*, sequence of characters composed from nucleic acids alphabet. The reason why it is comfortable to use DNA for data storage is the ease of mapping from the quaternary system to an array of characters(A, C, T, G) and vice versa. Each molecule of DNA is a double helix 2.1 formed from **two complementary** strands of nucleotides bonds(chemically hydrogen bonds) between G-C and A-T base pairs.

So in summary, *error control coders* consider single strand of DNA molecule - sequence, that is an ordered combination of 4 bases T,A,G,C, e.g.: ACCTGAT. Figure 2.2 demonstrates schematic representation of DNA molecule.



Figure 2.1.: DNA Molecule: double helix

2.3. Synthesis

Synthesis is a biological paradigm aimed to develop new *biological* components, tools, and structures, or to reinvent structures already present in nature [9]. Basically example of synthesis is to create synthetic molecules out of a certain sequence of DNA. In other words here, we are trying to create DNA physically from its sequence. **Synthesizer** a device which makes short, artificial oligonucleotides with any desired sequence of nucleotide bases. Let us decompose this definition.

Oligonucleotide is a sequence, that is an ordered combination of 4 bases T,A,G,C, e.g.: ACCTGAT .

- According to the definition of synthesizer, we pass the oligonucleotide sequence as an *input* to the system.

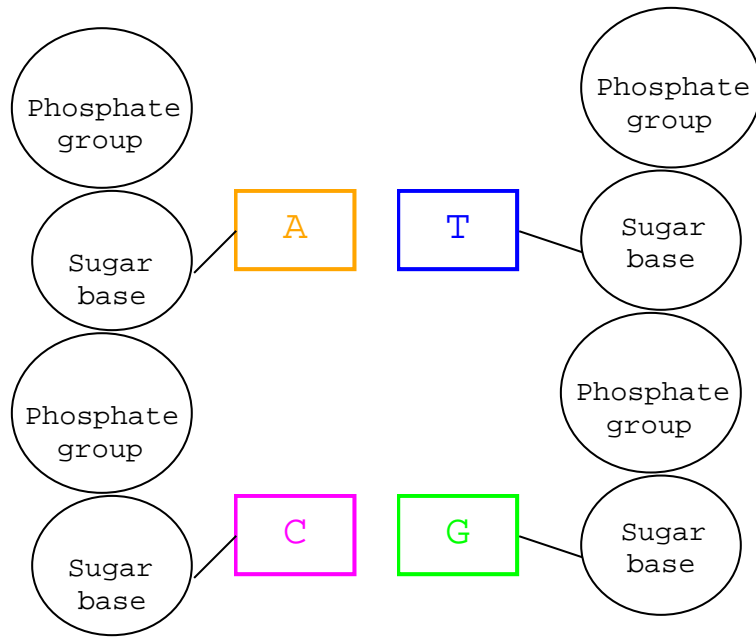


Figure 2.2.: DNA Structure: schematic visualization of DNA molecule

- In turn, using certain chemicals(phosphoramidite is a basis of reaction) machine creates artificial DNA molecules - by adding a complementary strand to the system. As a result, we get DNA molecules, that structurally are similar to the cell DNA molecule.

The only difference is cell DNA originates in the cell, synthetically produced DNA originates from phosphoramidite. Figure 2.3 depicts schematic representation of synthesizer.

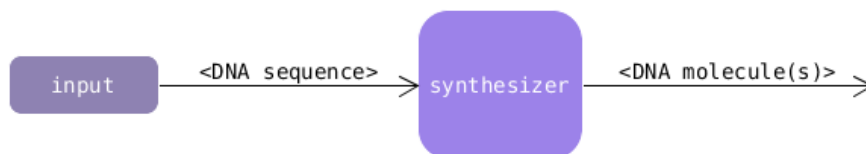


Figure 2.3.: Schematic representation of synthesizer

2.4. Sequencing

DNA sequencing is the process of determining the sequence of nucleotides (As,Cs,Gs and Ts) in piece of DNA [10]. Example of sequencing is to read the DNA sequence out of a certain collection of molecules. **Sequencer** is a device that automates the process of DNA

sequencing. The process behind the sequencing is both biological and computational. This chapter will concentrate on sequencer as a black box and in chapter three computational part will be discussed in detail.

Following list describes steps necessary to take in order to assemble genome [11](step-by-step schema is also shown 2.4).

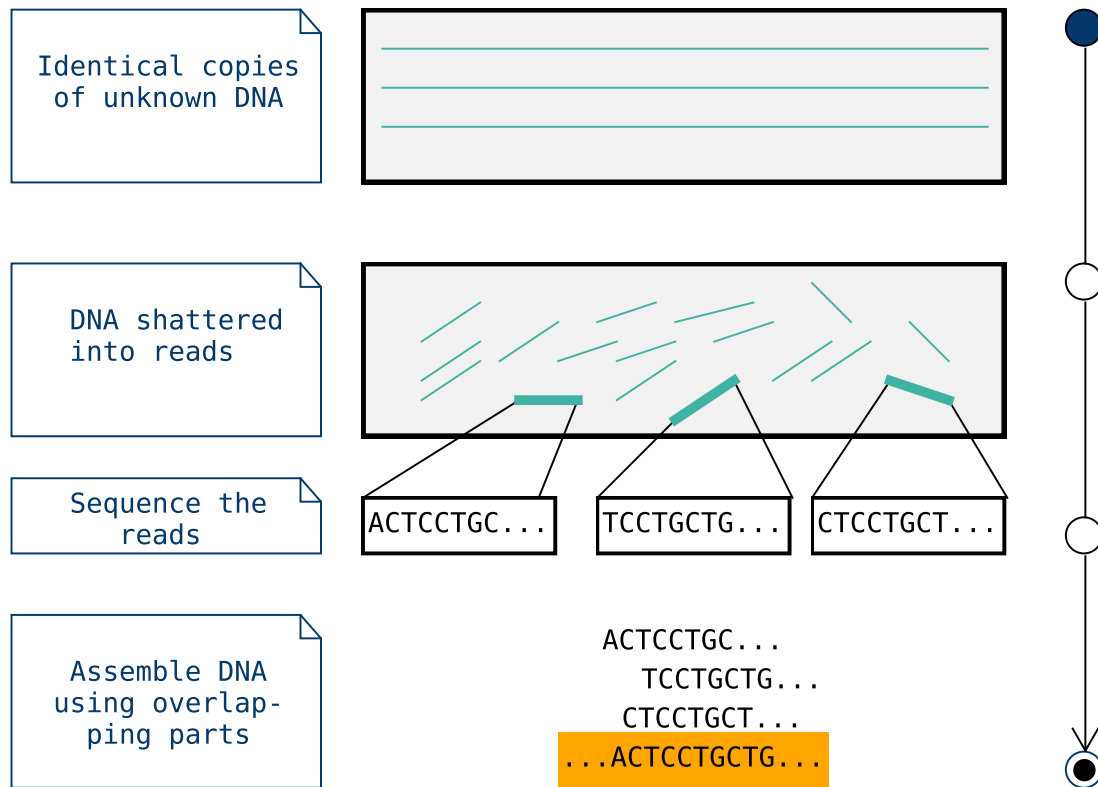


Figure 2.4.: Step-by-step representation of sequencer process. Assembly is multiple step process that has biological layer and computational layer. In biological layer scientists are collecting reads together; in computational layer using overlapping parts and techniques from graph theory the sequence is assembled.

- As an input sequencer takes a biological sample containing cells in our case molecules with certain DNA and creates multiple copies. Notice that goal of sequencer is to assemble the DNA sequence, so sequencer has no idea about the information stored in DNA.
- Using biochemical methods, the researchers are breaking DNA into fragments, and then sequence the fragments to produce **reads**. **Reads** are usually DNA sequences of length 20-300(20-40 are called ultrashort, 100-300 - long reads).

- Now that the reads are given algorithm assembles the reads into one sequence (please notice the order of the reads is unknown).

DNA assembly is one of the most challenging parts in bioinformatics, and the research is still ongoing.

2.5. Putting biological part altogether

From the previous subchapters it can be seen that synthetic biology can be used as a certain biochemical interface. We can consider **synthesizer** as an **input** for creating molecules and **sequencer** as an **output** for reading back the the DNA sequence from the molecule.

However, one component that an interface system lacks is certain physical medium, where DNA molecules are contained. State-of-the-art technologies at the moment suggest a certain jar with chemicals for storing molecules and call it **DNA soup** [12].

Let us put together the system that will serve as an interface. In the next chapters, we are going to incorporate channel coding into our system for completeness of DNA as storage. Figure 2.5 demonstrates synthetic biochemical interface.

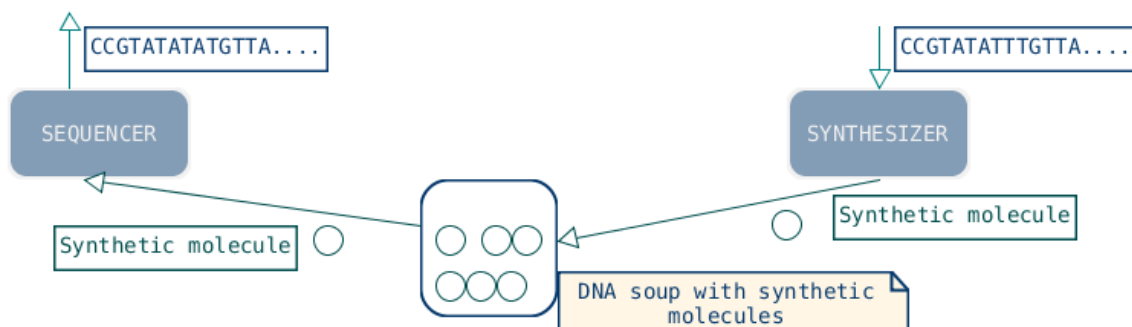


Figure 2.5.: Biochemical interface for DNA as a storage altogether

3. Error Control Codes

First chapter described biological interface of DNA as a storage work. This chapter will concentrate on error control codes area, which deals with data storage techniques. Techniques include error detection and correction.

3.1. Introduction

Coding theory is the process of converting information from a form convenient for direct use to a form convenient for transmission, storage, automatic processing and safety from unauthorized access. Several applications of coding theory exist cryptography, data compression, data storage and networking[13].

This chapter is going to concentrate on data storage. In order to proceed to the biochemical interface, the information must be **encoded** in a way that conveniently fits schema. Additionally, to the encoding process, it is necessary to establish the **mapping** schema between quaternary DNA alphabet and 0,1,2,3. Next section covers **finite fields** since in case of DNA alphabet is quaternary. Since the work is concentrated mainly on errors, that occur during sequencing, penultimate section concentrates on **detect the errors**.

3.2. Forward error correction

Forward error correction (FEC) - is a technique of noise-resistant coding and decoding, which allows correcting errors by the preemptive method. FEC is used to correct failures and errors during data transfer by transmitting redundant service information, on the basis of which the original content can be restored. The given system can correct one error and detect three errors [14]. Error-correcting code is a technique for adding redundancy to a message in a way such that it can be recovered using various mathematical methods. Mainly there are two types of error-correcting codes:

1. Algebraic Codes: Given information is split into small pieces, where each piece corresponds to a message. Messages are consequently encoded into codewords (blocks). Examples of algebraic codes are Reed-Solomon codes, Hamming codes, Golay codes, CRC codes and. These types of codes are based on Boolean algebra and also referred to as block codes.

2. Convolutional codes: here the codes may operate on potentially infinite sequences of elements and not on finite blocks. However, there is no specific algebraic theory behind the construction of convolutional codes. Examples of techniques used on convolutional codes are the Viterbi algorithm.

This work concentrates on **algebraic/block codes**.

3.3. Finite Fields and Mapping

Finite field is a mathematical term and has applications in many subjects such as control coding, cryptography and computer science [15]. Basically as a name suggests, it is a *finite set* of symbols that consists of its own arithmetic, specifically *addition* and *multiplication*, by following some defined rules. Addition and multiplication behave in finite fields similar to real and rational numbers; difference is that in former case addition multiplication are carried out by modulo. It is worth mentioning that finite field has additive and multiplicative neutral elements, which are normally 0 and 1.

Galois field is a finite field where the arithmetic is carried out by **modulo**. So basically, we construct the *addition* and *multiplication* tables, where the operation is carried out between corresponding elements. After the latter as soon as the value of element is greater than the number of field elements, e.g. size is prime, we determine the value by performing modulo operation - $\text{mod}(p)$. Quantity of elements in the finite field is its order(true only if). For example in \mathbb{F}_3 - field with 3 elements or $\text{mod}3$. Each element has additive and multiplicative inverses except for 1 and 0.

In case of DNA as a data storage we have quaternary system - alphabet of four characters, which will be mapped to $\{A, C, G, T\}$. In order to create such a system it is necessary to look at Galois fields. According to *Galois theory* any field of power q and prime p can be constructed if there is given an irreducible polynomial $p(x)$ of order q . In our case in order to construct \mathbb{F}_{p^q} , where $p = 2$ $q = 2$; one should use the following polynomial: $p_{2^2}(x) = x^2 + x + 1$. By using a given polynomial one can list all the possible elements (4 elements in a field) that will appear in a given field. As a consequence this polynomial will serve for a modulo operation as : $\text{mod}(p_4(x))$ while performing the operations like multiplication and addition. In fact all elements of this field will be constructed from a given polynomial and as a consequence we will have a field with 4 elements, that are listed below in list : $0 = 0$, $1 = 1$, $2 = x$, $3 = x+1$. Tables 3.1 and 3.2 demonstrate addition and multiplication operations in \mathbb{F}_4 .

3.4. Error Detection

The **error detection** step we cover here is going to be executed right before the computational procedure in sequencing, to detect the errors in reads - purify data. In this work we decided

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

Table 3.1.: Addition table

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

Table 3.2.: Multiplication table

to use **CRC codes** since they have specific properties that are comfortable to use correctly for DNA as storage.

A **cyclic redundancy check (CRC)** is an error-detecting code commonly used in digital networks and storage devices to detect accidental changes to raw data [16]. Usually, the data that is sequenced from reads has certain error probability, CRC codes will help us detect them. There is a particular mathematical procedure that needs to be done in order to achieve detection. We must first have the **generator matrix** and proceed to multiply each message with the latter.

Let us see step-by-step **encoding** and **error detection**. It is worth mentioning that CRC codes have the generator polynomials. Based on the example that is conducted on our simulation generator polynomial is

$$x^4 + x^3 + x^2 + 2x + 3.$$

and our code is $[\mathbb{F}_4; 15; 11]$ (n -codeword length, k -message length), where $n = 15$ and $k = 11$.

- Generator matrix from that polynomial and code parameters $[\mathbb{F}_4; 15; 11]$ is as follows:

$$\underline{\underline{\mathbf{G}}} = \begin{pmatrix} 1 & 1 & 1 & 2 & 3 & \dots & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 2 & 3 & \dots & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 2 & 3 & \dots & 0 \\ \vdots & 0 & 0 & \ddots & 0 & 0 & 0 & 0 & \vdots \\ 0 & \dots & 1 & 2 & 3 & 2 & 3 & 0 & 0 \\ 0 & 0 & \dots & 1 & 2 & 3 & 2 & 3 & 0 \\ 0 & 0 & 0 & \dots & 1 & 2 & 3 & 2 & 3 \end{pmatrix}$$

- Additionally for further processing, it is necessary to bring the matrix to reduced echelon

form in order to get the **systematic** generator matrix:

$$\underline{G} = [\underline{I}_k | \underline{P}],$$

where k is message length, I - $k \times k$ identity matrix and P - is **parity**.

- Now to encode each message into codeword following equation is used:

$$c_i = m_i * \underline{G},$$

where i serves as a counter element.

After performing procedure described above we have set of codewords- $\{c_1, c_2, \dots, c_i, \dots\}$. Now, suppose there has been distortion (in our case in reads might be erroneous) and erroneous codewords must be discarded. Procedure is as follows:

- First of all it is necessary to transform *systematic generator* matrix to **parity check matrix**. In order to do so, it is necessary to use following equation:

$$\underline{H} = [-\underline{P}^T | \underline{I}_{n-k}].$$

- Next we shall multiply each codeword from $\{c_1, c_2, \dots, c_i, \dots\}$ by parity check matrix:

$$\underline{s} = \underline{H} * \underline{c},$$

where s is a **syndrome**. One of two possibilities happen:

- $s=0_v \Rightarrow c$ is indeed a codeword;
- $s \neq 0_v \Rightarrow c$ is not a codeword.

This work is mainly concentrated on detecting the erroneous codewords and discarding them for further processing(sequencing and decoding).

3.5. Putting error control codes altogether

From previous subsections we have gathered enough knowledge to put the error control coding system together. Figure 3.1 displays our system in more details. Process of storage is as follows:

- Source contains certain payload, let us say e.g. set of messages.
- They set of messages - $m = (m_1, m_2, m_3, \dots, m_i)$ gets encoded into codewords $c = (c_1, c_2, c_3, \dots, c_i)$ through **encoder** using techniques described in previous subsection.
- Codewords $c = (c_1, c_2, c_3, \dots, c_i)$ are passed to biochemical interface (ref. chapter one), where they are distorted; we might think of biochemical interface as a **channel**.

- Sequenced data is mapped from DNA alphabet(A, C, G, T s) to \mathbb{F}_4 and is passed through **decoder**, where codewords $c = (c_1, c_2, c_3, \dots, c_i)$ are decoded to message estimate $m' = (m_1, m_2, m_3, \dots, m_i)$.

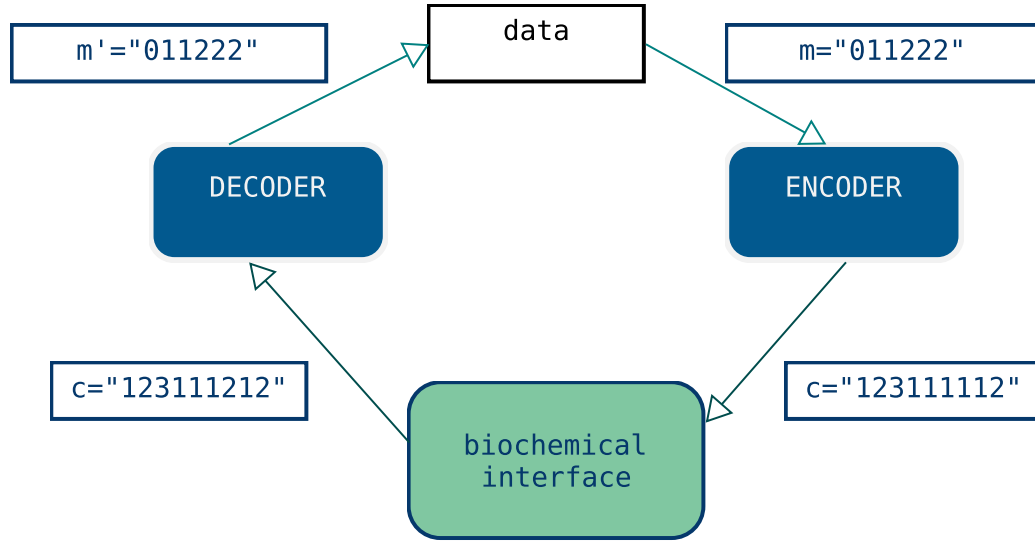


Figure 3.1.: Basic workflow of DNA as a data from Error Control Codes perspective. Here m is a message vector, c is codeword and m' is message estimate.

Technique described above is useful for delivery of information through unreliable channels.

Errors and erasures in the DNA system are described in the list below [17]:

- Either molecule is not synthesized at all, or it is synthesized more often than others(usually each molecule is synthesized once). Contemporary synthesizers typically generate multiple thousands of copies of the chain, which might contain several kinds of errors.
- Molecules in DNA soup itself might decompose, which results in loss of information/-molecules.
- When the molecules are extracted from the jar, depending on distribution, the only fraction is sequenced.
- Sequencing DNA drives to nucleotide insertion, deletion or substitution in particular DNA molecules.

This work tackles the type of errors that appear during sequencing and introduces coding scheme in between biological and computational layers.

4. Workflow of DNA as a storage

Chapters before have covered main building blocks of this work: error control codes and bioinformatics. From error control coding point of view, the main components are encoder, channel and decoder. From a biological point of view, the main components are synthesizer, DNA soup/channel and sequencer. In this chapter, we are going to cover the whole system taking into account both areas.

Additionally, the package structure of the simulation/setup and computational layer in the sequencing problem are discussed in this chapter. Additional layer in between biological and computational parts of the assembly problem is introduced. As chapter describes a new layer will instigate new problems.

4.1. Schema of DNA as a storage

The best way to display the whole DNA as a storage system is to visualize it. Figure 4.1 displays the whole process. Let us go through once more through the main parts of system and the parts this work is going to concentrate on. Figure 4.1 has step numbers around each component, which correspond to the list below in process [18]:

1. We have certain payload - data and we decompose it in \mathbb{F}_4 type messages - $m = (m_1, m_2, m_3, \dots, m_i)$.
2. In this step we **encode** the payload m into $c = (c_1, c_2, c_3, \dots, c_i)$.
3. In order to synthesize the codewords we have to first **map** it to quaternary alphabet - $\{A, C, G, T\}$. In this step, our approach is also to concatenate the codewords together into one sequence and then pass them to the synthesizer.
4. After concatenating codewords into one sequence, we **synthesize** the sequence into actual molecule/molecules (depending on the machines used in the process).
5. This is the illustration of a jar, the so-called **DNA soup**, where synthetic DNA molecules are floating around; however, the order of molecules is undefined. Please note here DNA breakage, which is equivalent to symbol/block deletion or even losses of whole sequences might happen!

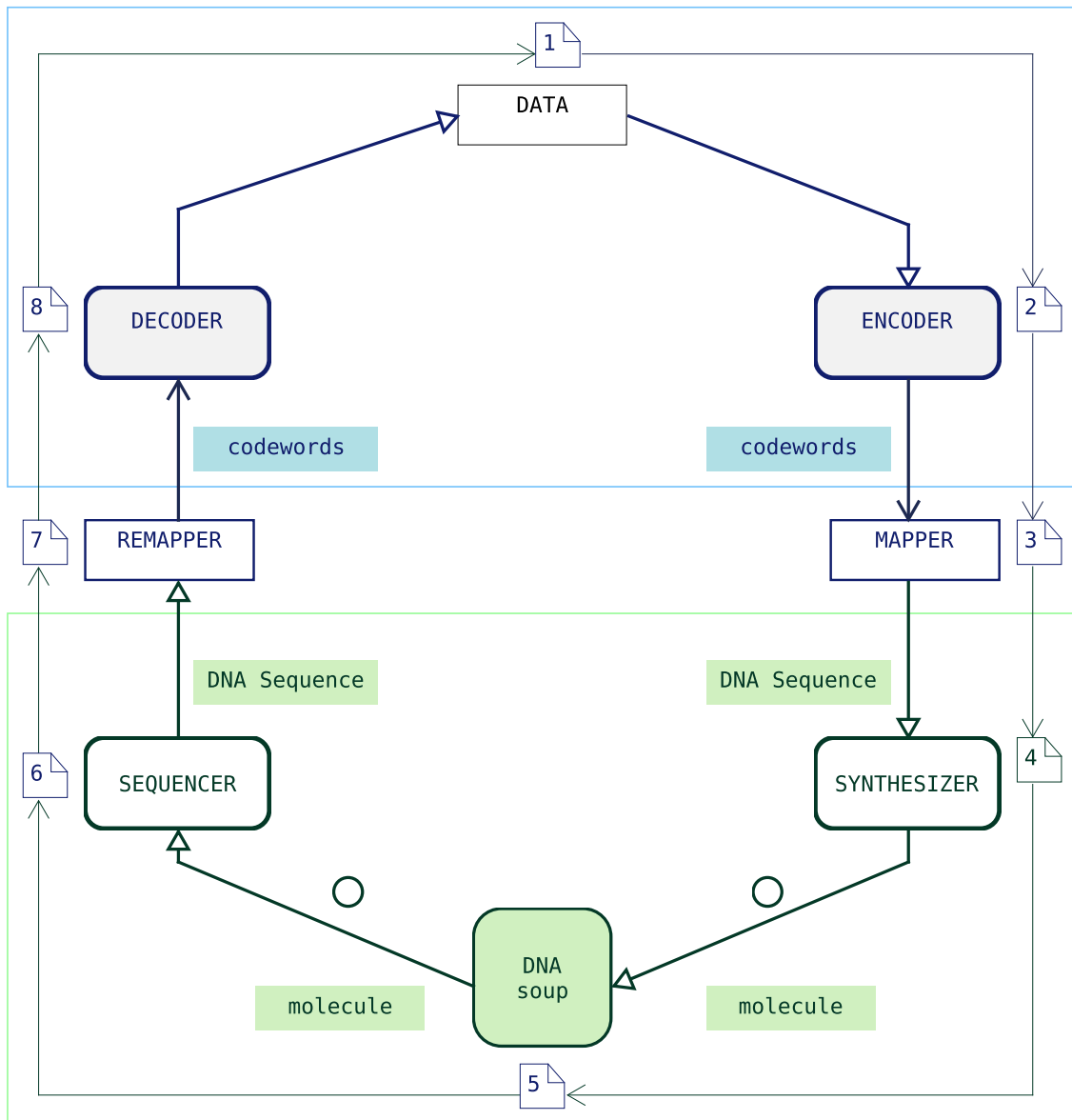


Figure 4.1.: Basic workflow of DNA as a data from both biological and ECC perspectives. As it was described before - messages stands for set of messages, codewords for set of codewords, and DNA soup is jar with encoded molecules.

6. Molecule(s) is picked randomly uniformly and passed to be **sequenced**. Please note this is the point where the errors/insertions/deletions or substitutions may happen. Here error control codes detect erroneous reads.
7. Here we **re-map** the sequenced data - which is DNA sequence back into \mathbb{F}_4 in order to be passed to decoder.
8. Finally we decode the codewords into messages, where we get the message estimate $m' = (m_1, m_2, m_3, \dots, m_i)$.

Errors in strands appear in synthesis, storage(to a smaller extent due to decay of DNA) and sequencing steps. Synthesis adds largely deletion and likely insertions. Given that NGS (next-generation sequencing) technology is used for gathering DNA strand, step 6 includes substitution and very few insertions or deletions.

The goal of this work is to introduce a particular way of minimizing errors that occur during step 6 of the process in a sequencer state. The issues that occur during sequencing are erroneous reads that cannot be reconstructed using one of the classical algorithms for assembly. Even though there are specific correction approaches for the assembly, they are heavily heuristic. We decided to approach the problem as error control coders by introducing intermediate steps in the beginning and in between sequencing steps.

In the next two sections, package structure and sequencing problem are going to be discussed. Third and fourth sections of this chapter will explain sequencing problem in greater detail, including algorithms, since it is necessary to understand the way assembly process works computationally and why there is a necessity for detecting the errors.

4.2. Tools used for DNA as a storage

As a programming language in this work, we decided to utilize python for two reasons: ease of use and possibility to reuse algorithms for bioinformatics. This section gives a brief description of **structural viewpoint** or in other words, class diagrams of setup for simulation. Nevertheless, before it is necessary to understand the philosophy of our approach to simulation.

- **Python:** As a programming language Python was a choice. Python is dynamically typed PL and has the garbage-collection capability. Furthermore, it supports various paradigms, such as object-oriented and functional programming [19].
- **UML2.0:** The most important reason why UML2.0 has been chosen as a tool is the ability to maintain a complex system with the help of structured documentation and capability to translate at least the structure to any object-oriented programming language [20]. UML 2.0 has viewpoints and views for modelling the software [21]. Viewpoints

are structural or behavioural entities (in a given context viewpoint are types of diagrams) that encompass the views, where UML 2.0 diagrams represent the latter.

- **OOP**: Object-oriented programming (OOP) is a concept that is designed to facilitate the development of complex systems by introducing new concepts that are closer to the real world than functional and procedural programming languages [22]. In the case of DNA as a storage system, as shown in Figure 4.1 is component-based. The usage of OOP, in this case, helps organize components as classes. Each of those classes is going to have a certain set of properties and methods. Classes are going to be low coupled (degree of binding between classes is low) and highly-cohesive (each of those classes has one specific goal). Additionally, one controller class is managing the workflow of DNA storage is going to be appended to the overall system.

All the schemas and models in the next subsections are represented as **class diagrams** from a structural viewpoint. A detailed description of the methods used in the system and corresponding classes are provided in the appendix.

4.2.1. Error control codes package

For error control codes package we have two classes: **Encoder** and **Detector**(since our Channel is special, we put it in helpers package) as shown in figure 4.2. Each one of those classes has corresponding methods that serve specific purpose; every method will be described in corresponding figure (for detailed overview please refer to code listings in appendix).

- **Encoder** - contains three methods: `construct_generator_matrix()`, `encode` and `encode_messages`. Description of methods and attributes is shown in figure 4.2.
- **Detector** - contains three methods: `check_whether_codeword()`, `get_h_matrix()` and `perform_calculation_to_checks`. Description of methods and attributes is shown in figure 4.2.

4.2.2. Bioinformatics package

For bioinformatics package we have two classes: **Sequencer** and **Synthesizer** as shown in figure 4.3. Each one of those classes has corresponding methods that serve synthesis and sequencing purpose - in other words biological layer; every method will be described in corresponding figure (for detailed overview please refer to code listings in appendix).

- **Synthesizer** - contains two methods: `join_for_synthesis()` and `map_codewords()`. Description of methods and attributes is shown in figure 4.3.
- **Sequencer** - contains three methods: `remap_codewords()`, `create_reads()` and `shortest_common_superstring_assembly()`. Description of methods and attributes is shown in figure 4.3.

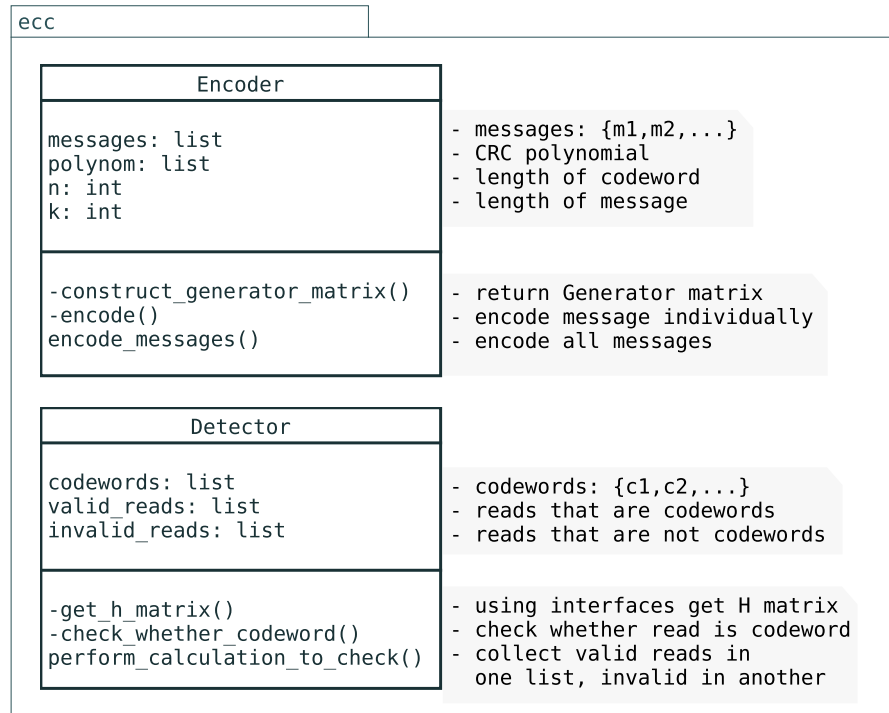


Figure 4.2.: Error control codes package; contains two classes only: Encoder and Detector.

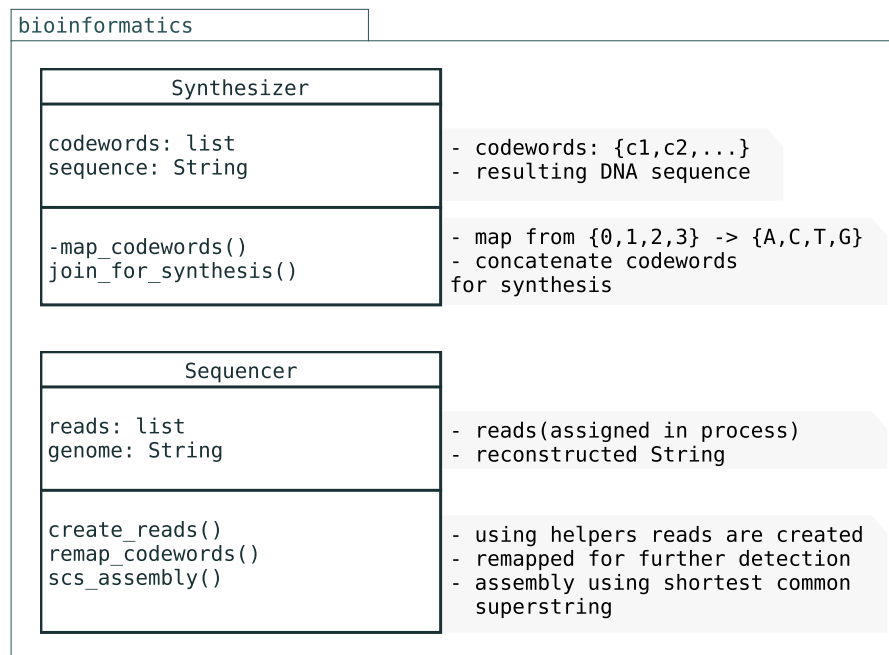


Figure 4.3.: Bioinformatics package; contains two classes only: Synthesizer and Sequencer.

4.2.3. Helpers package

One more package worth mentioning is **helpers** package, where all the necessary intermediate interfaces are stored. Intermediate interfaces and classes include \mathbb{F}_4 , **Mapper** and **Channel**, that imitates our distortion: where character selected at a random position gets distorted. Last but not least since our math is executed in \mathbb{F}_4 , operations are prototyped in *math_helpers.py* file - it contains methods necessary to execute e.g. matrix_vector multiplication in \mathbb{F}_4 . Figure 4.4 describes helpers folder.

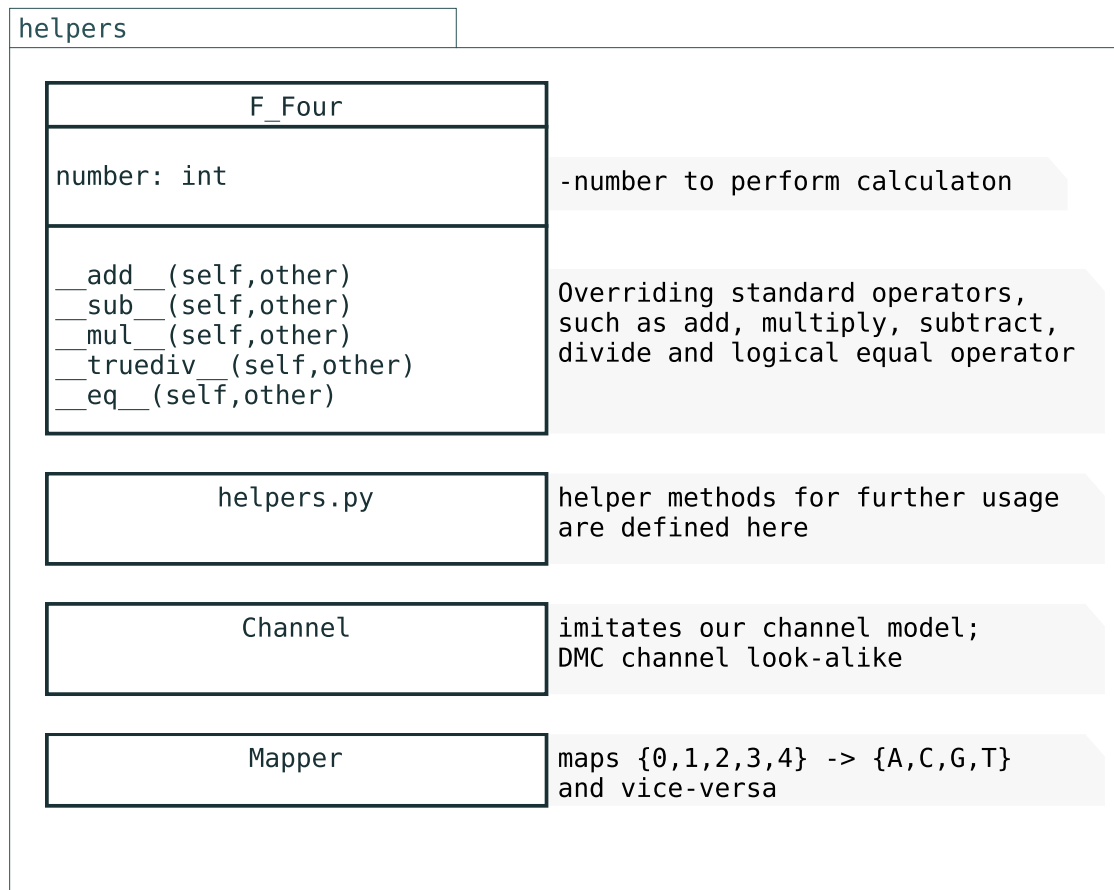


Figure 4.4.: Helpers package; contains three classes and one file: Mapper, Channel, F_Four and *math_helpers.py*.

4.2.4. Workflow manager

WorkflowManager is the main class, where the whole process is being executed. Workflow-Manager encompasses all the components described above in one manager class where it is

easy to control the system and where all those classes are mapped to individual methods. According to main principles from object-oriented programming our components/classes must be highly cohesive, low coupled. Each method demonstrated in figure 4.5 is responsible for one task displayed in our previous schema. WorkflowManager was designed, taking into account those tasks and principles and is shown on 4.5.

WorkflowManager	
<pre> messages: list polynom: String n: int k: int ENCODER: Encoder SYNTHESIZER: Synthesizer CHANNEL: Channel DETECTOR: CodewordDetector SEQUENCER: Sequencer </pre>	<ul style="list-style-type: none"> - messages: {m1,m2,...} - polynom - length of codeword - length of message - Encoder from ecc package - Synthesizer from bio package - Channel from helpers package - Detector from ecc package - Sequencer from
<pre> encode() map_and_synthesize() distort() bio_layer_sequence() ecc_layer_sequence() cs_layer_sequence() </pre>	<ul style="list-style-type: none"> - encode messages - map to DNA and concatenate codewords - distort codewords - make reads out of sequence - detect erroneous reads - assemble reads together to one String

Figure 4.5.: WorkflowManager class; manages all necessary components using packages described above.

4.3. Sequencing as a computational problem

Suppose there is a DNA sample which should be sequenced. As it was described in chapter [1] in order to sequence certain DNA sample, researchers first shatter it into multiple pieces - **reads**. They do so because biologically it is tough to sequence the long DNA strand - certain techniques used in the labs are still limited, and there is much research on that topic. Biologists sequence reads and collect them in a certain database (normally they are plain text files, e.g., *.fasta*, *.fastq*) [23]. Now it is up to computer scientists to collect the reads back, and output sequenced DNA. Before looking at the algorithms for sequencing, it is required to understand the data/reads saved in a database.

4.3.1. Exploding newspaper

To simplify the sequencing algorithm and problem even further bioinformatician Pavel Pevzner associated it with exploding newspapers problem [24]. Everything described here is purely theoretical and serves for the explanation. Illustrative depiction of the problem is displayed in figure 4.6.

Illustration is taken from: Phillip Compeau, Pavel Pevzner - Bioinformatics Algorithms: An Active Learning Approach

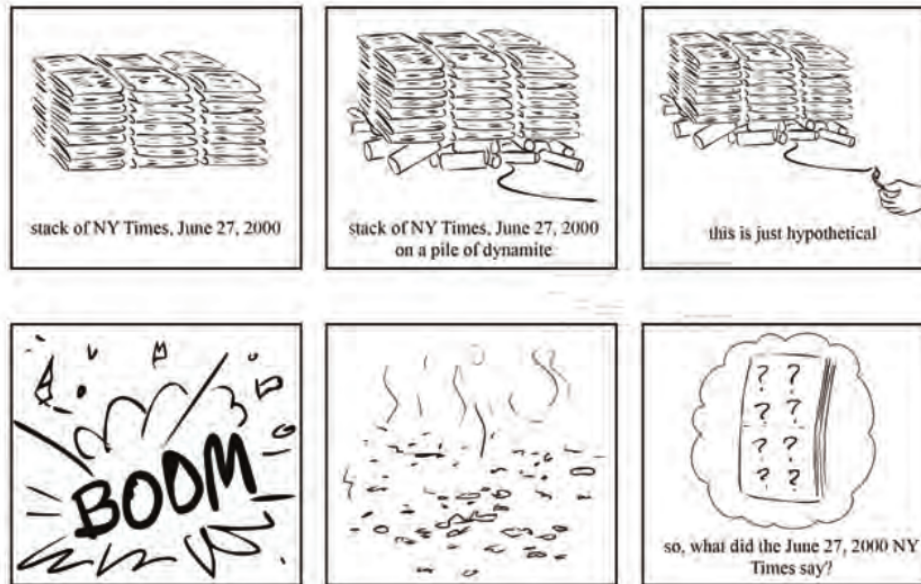


Figure 4.6.: Exploding newspaper problem: On the top left it is seen that hundreds of copies of the newspaper are stacked. In the top middle placed under dynamite. Out of pieces, we must recollect the picture shown below on the right.

On the top left, there are hundreds of copies of the newspaper. Next, we place dynamite and start the fire. The explosion proceeds right after setting dynamite on fire, and then there are parts of all of the copies on the ground. The task of exploding newspaper is to recollect one copy of the newspaper. This process describes our DNA sequencing problem: parts of the text are our reads and newspaper is DNA strand.

Now the question is: how to go from little pieces of newspaper to one full copy? Answer to that question is simple: **overlapping parts** as shown on figure 4.7 . As it is depicted certain parts of the text in newspaper are overlapping, e.g. *yet named* or *is welcomed*.

When the DNA sample is shattered we get the very similar picture: if there are enough shattered parts the DNA sequence or **genome** can be recollect.

Illustration is taken from: Phillip Compeau, Pavel Pevzner-Bioinformatics Algorithms: An Active Learning Approach

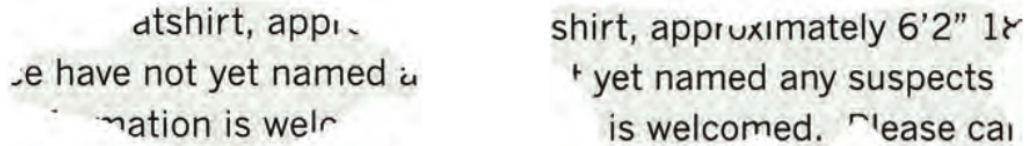


Figure 4.7.: Exploding newspaper problem: single pair of overlapping parts. Newspaper is being recollected from multiple pairs of the parts and as displayed if there are enough parts we can recollect the copy accordingly.

The task is not as simple as it seems since we have to take into account that reads are chaotically distributed when passed through an algorithm. Determining the order of nucleotides in a genome, or genome sequencing presents a fundamental task in bioinformatics. There are certain algorithms, specifically graph algorithms that are responsible for that.

4.3.2. Shortest common superstring and de Bruijn graph

As it was outlined in previous subchapter our data is very much alike the pieces of newspaper with many copies. There are many algorithms on this topic and it is a major research area. Sequencing the genome from those reads is called **genome assembly** problem. Before going into graph algorithms used in this work we must first note that CS researchers decided to shatter those reads further into **k** pieces and called them *k-mers* as shown on figure 4.8. However in our case since they are not very long and modern sequencers can output the reads with same length we will not shatter them further and call them *n-mers*.

A little refresher on definitions for graph algorithms for definitions is shown in 4.9. Algorithms that we are going to concentrate on are **SCS - shortest common superstring** and **de Bruijn assembly**:

- **SCS**: Formulation of shortest common superstring [25] is shown in figure 4.10: Researchers decided to approach this problem using **directed overlap graphs**. As it was described above we must base our algorithms on overlapping parts, so overlap graph is one of those approaches. Furthermore it is necessary to find the shortest path in that overlap graph, depending on whether *n-mers* have enough overlapping parts. Additionally, we assume that each **vertex** of overlap graph is actually an *n-mer*. Sometimes it is referred to as **travelling salesman** problem, where one should find **Hamiltonian path** [26].

Definition 4.1 A *Hamiltonian path* is a path that passes through every vertex exactly once.

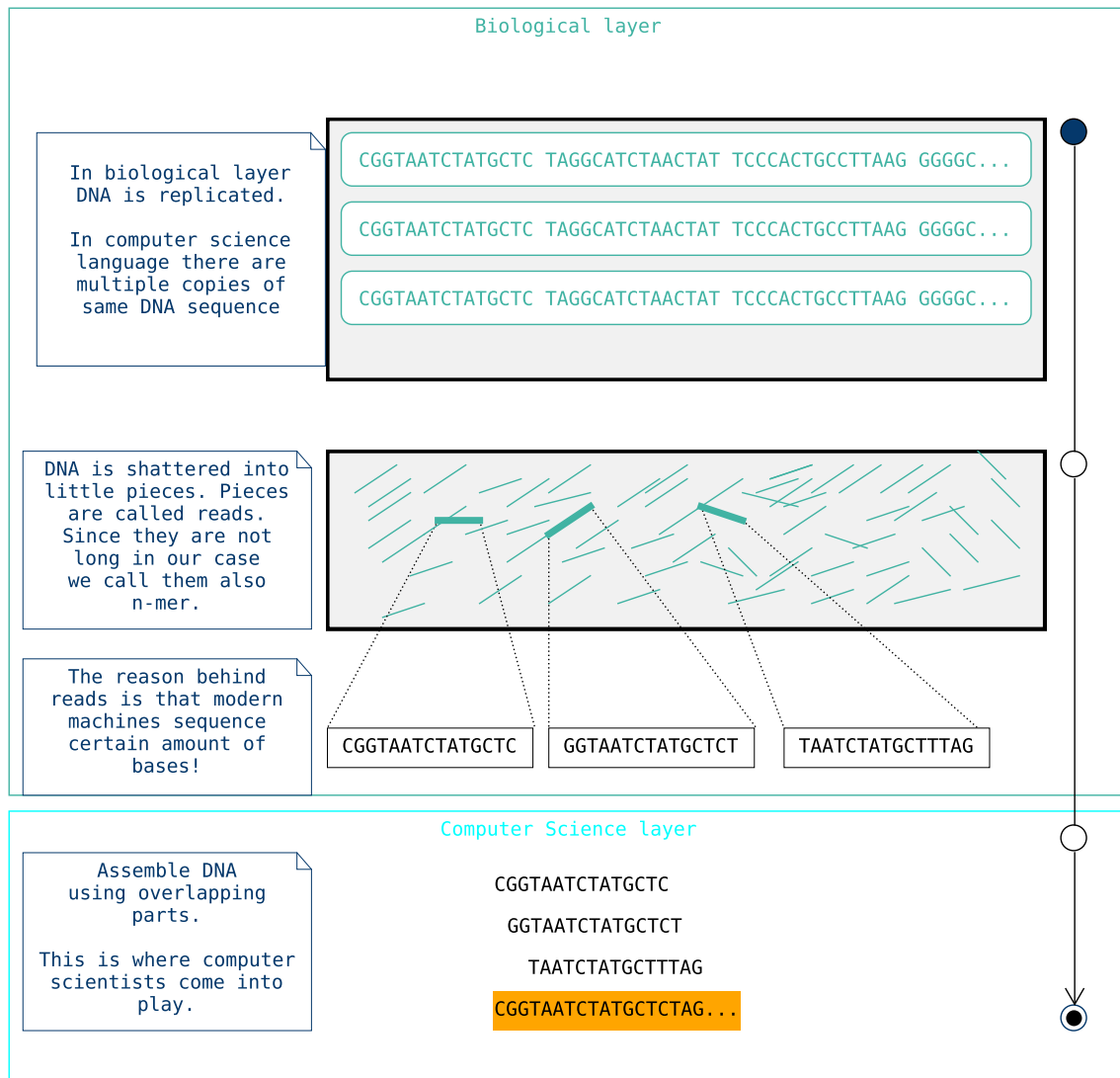


Figure 4.8.: Sequencing explained in detail. As it can be seen on figure in biological layer we sequence reads(shattered parts) individually and save them to .fastq or .fasta file. Afterwards we pass it to one of the assembly algorithms where they are collected into one string.

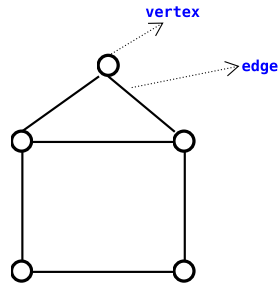


Figure 4.9.: Definitions from graph theory.

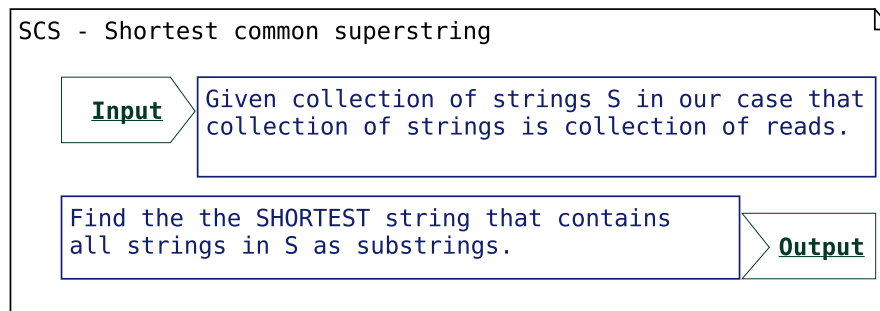


Figure 4.10.: SCS algorithm formulation

However the problem is **NP-hard** [27], so we will optimize the performance by minimizing the accuracy and using **greedy** version of SCS.

Greedy SCS algorithm is as follows (approach to solution is taken from *prof. Ben Langmead's slides*) :

1. `overlap(a, b, min_length=3)`: Return length of longest suffix of `a` matching a prefix of `b` that is at least `min_length` characters long. If no such overlap exists, return 0.
 2. `pick_maximal_overlap(reads, k)`: Return a pair of reads from the list with a maximal suffix/prefix overlap $\geq k$. Returns overlap length 0 if there are no such overlaps.
 3. `greedy_scs(reads, k)`: Greedy shortest-common-superstring merge method-repeat until no edges (overlaps of length $\geq k$) remain.
- **de Bruijn assembly**: De Bruijn assembly is very similar to the shortest common superstring. Formulation of the problem is the same as for SCS (see figure 4.10) [28]. Difference between SCS and de Bruijn assembly is that in latter case we must find **Eulerian** path [29].

Definition 4.2 An *Eulerian path* is a path that passes through every edge exactly once. If it ends at the initial vertex then it is an *Euler cycle*.

Difference between these definitions is that one of them covers all the vertices, the other covers all the edges! Condensed version of the algorithm's pseudocode is as follows:

1. `dB := DeBruijnGraphGenerator(reads)` - generate de Bruijn graph where each vertex is $n - 1$ - mer and each edge is n - mer.
2. `path := EulerianPath(dB)` - find Eulerian path in generated de Bruijn graph.
3. `reconstructedString := PathToDNASequence(path)` - method that converts path to DNA sequence.

The algorithms presented above work with certain precision and depending on the length of overlaps and repetitiveness of certain reads in genome the outcome may alter.

4.4. Error detection for sequenced reads

In the above sections, we have seen how the DNA as a storage system is organized as a whole and what are concrete algorithms for sequencing. We have also investigated each component in previous chapters. In this chapter we are going to introduce additional helper interface - the coding scheme for sequencing.

Current state of the art technologies does not have a particular way of correcting errors in detecting erroneous reads. They just go through sequencer-generated data, count the most frequently occurring patterns and assume latter is erroneous.

Our approach is to introduce **error detection layer** in between biological and computational layers. Figure 4.11 demonstrates the intermediate layer introduced for sequencing process. The idea for detecting erroneous reads is to set the length of the reads to the length of codewords and later detect the erroneous reads using channel coding and error detection techniques.

If we think about the reads in terms of codewords then we may detect erroneous reads using coding techniques introduced in chapter two. However, we will lose overlapping codewords and algorithms - SCS and de Bruijn assembly will both fail to assemble the codewords. Let us think of the statement made above in terms of example.

- Given collection of seven messages $m = (m_1, m_2, m_3, \dots, m_7)$ and say $k = 11$.
- After proper encoding we end up with corresponding collection of seven codewords $c = (c_1, c_2, c_3, \dots, c_7)$, with $n = 15$. Notice we have only *seven codewords* each of length 15.

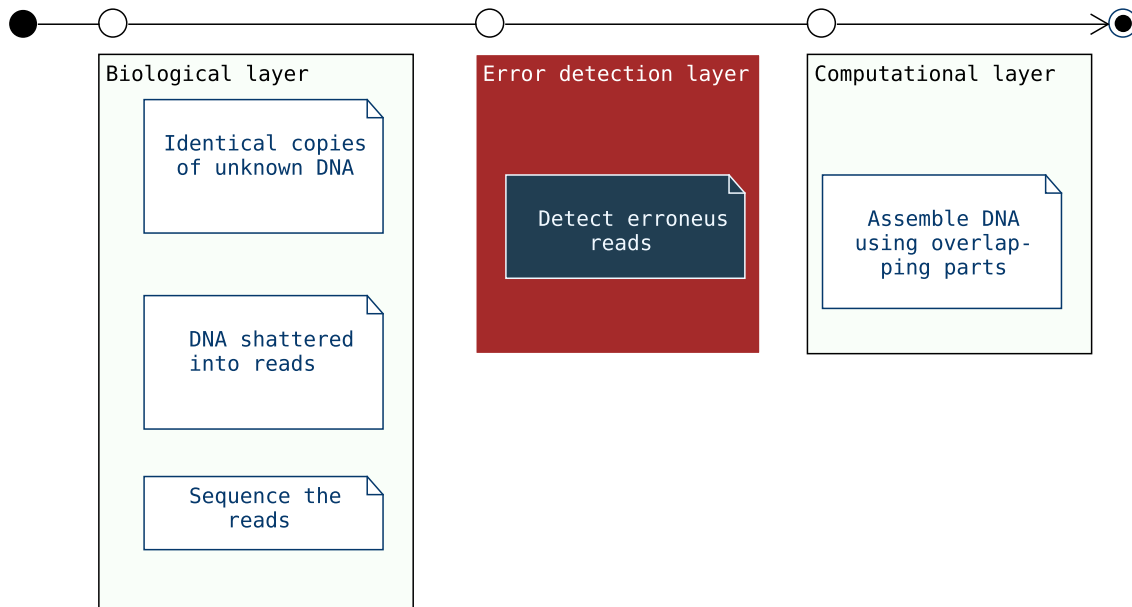


Figure 4.11.: Depiction of intermediate layer introduced for sequencing process in DNA as storage. Reads are being identified to be erroneous - discard the erroneous reads and keep the correct reads.

- We map those codewords to DNA, concatenate them and synthesize them. Length of full synthetic **genome** will be $7 * 15 = 105$; in other words, it is necessary to reconstruct a full synthetic sequence of length 105.
- Now on sequencer side, we have a collection of reads with length $n = 15$, the unknown genome of length 105 and a way to detect whether reads are **erroneus** or not. Nevertheless, there occurs a problem, which is demonstrated in figure 4.12.
- As it can be seen on figure 4.12 ideally reads are assembled as shown on the left part of the figure, such that when fed into one of the assembly algorithms, it can reconstruct the genome.
- However what happens in our case is that every 15th read is codeword and we end up with an insufficient amount of information. It causes problems in both graph algorithms introduced in the sections above.

To investigate the problem further let us consider basic sequencing setup where reads are distributed in a way normally required for de Bruijn graph or shortest common superstring. The way those algorithms assemble reads together is by "*gluing*" reads using overlaps. Basic terminology for the overlaps and the codewords are shown on 4.13.

In order to connect two pieces together, **suffix** of one read must be equal to **prefix** of the

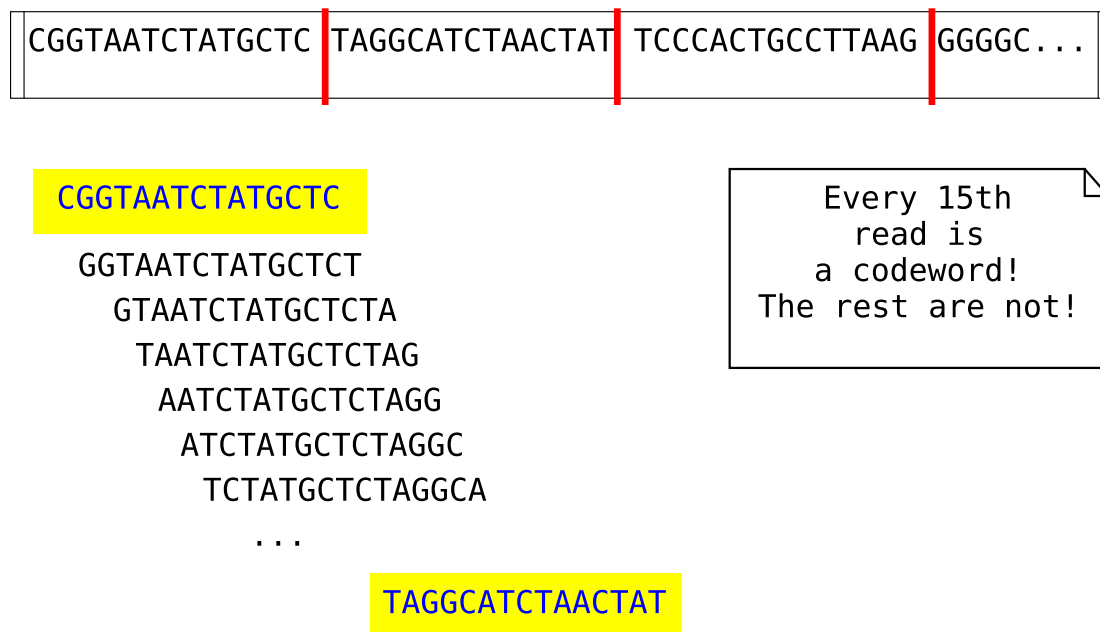


Figure 4.12.: Genome assembly from reads vs codewords. The figure demonstrates problems introduced by error control codes while discarding the possible correct reads.

next one. According to example shown on Figure 4.13 if we completely assign reads to codewords then all overlapping parts (from our initial setup only 7 codewords are given, which are concatenated together), because they were not transferred at first place. SCS algorithm is based on overlapping parts between the reads. Given there are no overlaps between *c1* and *c2* it is impossible to reconstruct the genome. As a conclusion, it is not possible for SCS algorithm to reconstruct the genome out of codewords only.

Next chapter addresses how this problem may be solved by adding redundancy in front of original messages for providing enough overlap to be detected. Additionally, it will be required to make certain modifications to original simulation to adjust to a new setup.

Since the simulation mechanism of the system is based on main OOP principles it will be straightforward to incorporate changes, only by adding few goal-oriented methods to controller class - *WorkflowManager*. For random data generation, it is necessary to incorporate an additionally *data.generator* method, where data is randomly generated.

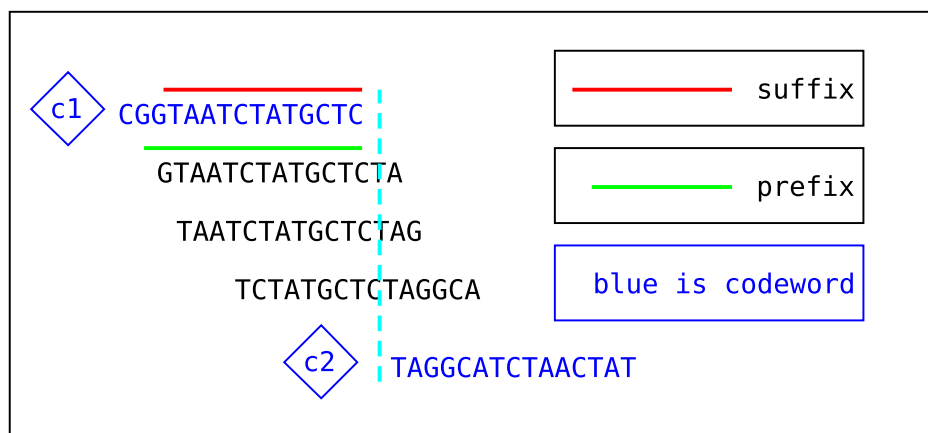


Figure 4.13.: Basic terminology in sequencing: **suffix** and **prefix**. Figure also demonstrates why it is impossible to recollect genome from **codewords** only. Please note there is no overlap between the c1 and c2, if we disregard the rest of reads from our system.

5. Workflow of DNA as a storage

In the last chapter we introduced an intermediate **error detection** layer between **biological** and **computational** layers in order to detect erroneous reads. However, the insufficient amount of reads (as it was agreed to make them the same length as codewords) is a major issue in the system.

As it was discussed in the previous chapter two algorithms are used for DNA assembly: shortest common superstring and de Bruijn assembly. The primary driving variable for those algorithms is based on **overlap** and the proposed codeword - read approach only took every n^{th} read as a codeword. In this chapter, the read-codeword difference and similar concepts are going to be explained in greater detail. Additionally, a new method of appending the redundancy to the beginning of the message is going to be introduced.

5.1. Reads vs. Codewords

This section will revisit the reads as a codewords concept. As it is shown in Figure 5.1 reads have certain overlapping parts; codewords have no overlaps at all. Codewords appear at every n^{th} position with a guarantee of no error. The problem in this case is that assembly algorithm is not going to work since no overlap is preserved.

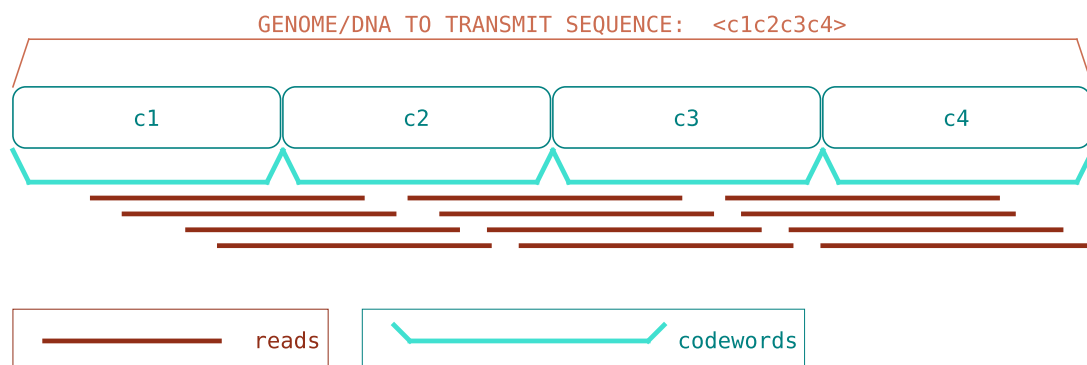


Figure 5.1.: Illustration of the fundamental difference between codewords and reads. The DNA sequence is a concatenation of all codewords.

Even though the DNA sequence consists of all **codewords concatenated together**, it is still unknown in which order they have been transmitted/stored due to the type of channel (all chemicals in one jar). That is the reason why reads must be in a structured manner that involves particular overlapping schema as it is shown in state-of-the-art sequencing technologies. On the other hand, it is known that sequencers are not very accurate, and errors may occur during the sequencing process in reads. For example, one of the most prominent situations that may happen is incorrectly identified symbols in the biological layer, which results in a sequence having erroneous reads.

The approach to the given problem in this thesis is introducing a header to the beginning of the actual payload. By adding a certain static header of length s we guarantee that there are at least s overlapping parts and at the same time the reads are not erroneous. By exploiting the shifting property of **cyclic codes**, some overlap will be preserved. In the next chapter, we will introduce the concept of **header**-which is a static sequence at the beginning of the payload, and **proto messages**-which will, in turn, be our payload.

5.2. Concept of headers and proto messages

The section will introduce the approach to sequencing the reads without errors; the explanation is based on a toy parameterized example as the initial setup.

Given a DNA as a storage setup with four messages of length 5 and CRC polynomial with corresponding generator matrix G . The list of all necessary steps to review in conventional (current state-of-the-art) DNA as storage system is as follows:

- Messages $m = \{m_1, m_2, m_3, m_4\}$ are transferred to encoder. As it can be seen in Figure 5.2 the length of each message is 7, e.g. $m_2 = [m_{21}, m_{22}, m_{23}, m_{24}, m_{25}, m_{26}, m_{27}]$.

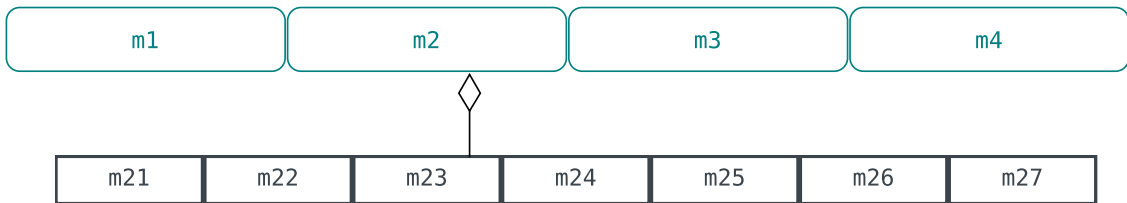


Figure 5.2.: Four messages that consist of five symbols each.

- Messages get encoded using systematic encoder (systematic $\underline{\underline{G}}$ matrix) and set of code-words $c = \{c_1, c_2, c_3, c_4\}$ with length 9 is generated.

- Let us closely inspect codewords transferred in our schema. As it can be seen in Figure 5.3, each codeword consists of 9 symbols $c_2 = [c_{21}, c_{22}, c_{23}, c_{24}, c_{25}, c_{26}, c_{27}, c_{28}, c_{29}]$.
- But what if one of the symbols is shifted, e.g., $c_2 = [c_{29}, c_{22}, c_{23}, c_{24}, c_{25}, c_{26}, c_{27}, c_{28}, c_{21}]$? Due to shifting property of cyclic codes used in this work, modified version of c_2 would still be recognized as a codeword.
- **Idea 5.1** By exploiting the shifting property of cyclic codes it is possible to organize the codewords in such a way that detector recognizes overlaps of certain length as valid codewords and preserve them for later assembly.

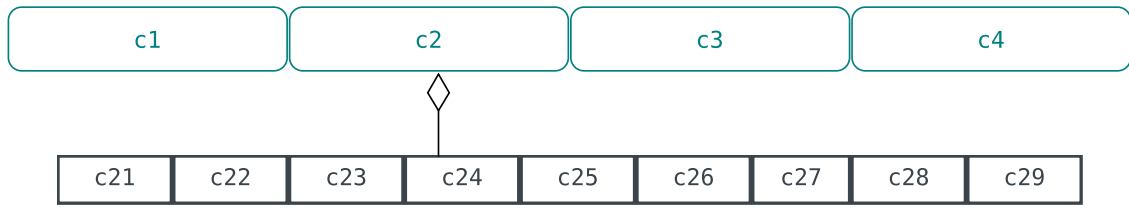


Figure 5.3.: Four codewords each consists of seven symbols.

- Let us adjust the setup to the proposed idea: we call original payload(information to be transferred) is updated to **proto messages** and the **header** is added to the beginning of our proto messages. The updated schema of messages is shown in Figure 5.4.
- **Header** is a statically defined sequence of length s that is concatenated in **front** of the payload.
- Message length - k is equal to

$$k = s + k',$$

where s is **header** length and k' is length of **proto messages**.

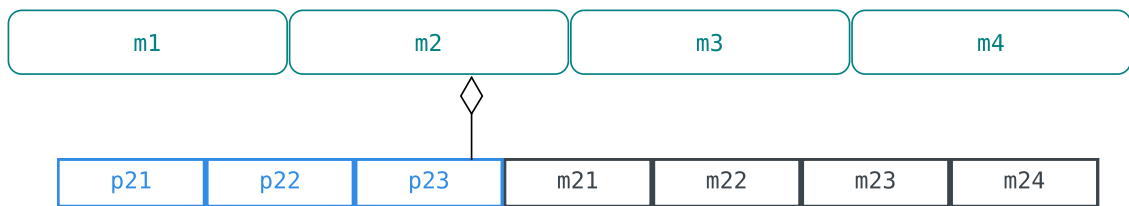


Figure 5.4.: The first three symbols are **header** and last four symbols are **proto messages**. As it can be seen now message is $m_2 = [p_1, p_2, p_3, m_{21}, m_{22}, m_{23}, m_{24}]$, where $s = 3$, $l = 4$ and $k = 7$.

- In the case of the concrete example $s = 3$, $k' = 4$ and $k = 7$. Three symbols in the beginning provide enough overlap in order to assemble the DNA sequence.
- Given that messages are encoded using CRC polynomial and dimensions of generator matrix are 7×9 , resulting codewords will be of length 9. Schema of resulting codewords is shown in Figure 5.5.
- Using strategy described above codeword length - n is

$$n = s + n',$$

where s is **header** length. For concrete example as a result we get $s = 3$, $n' = 6$ and $n = 9$.

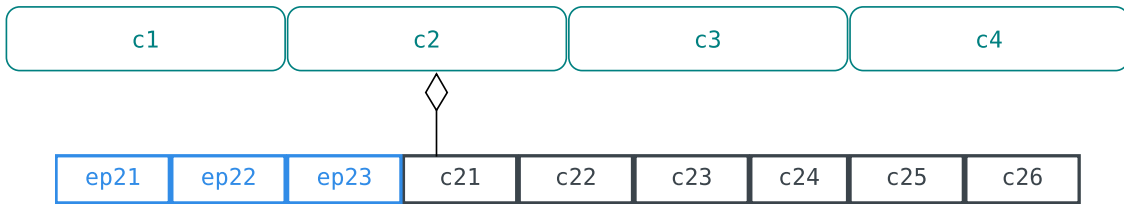


Figure 5.5.: The first three symbols are **header** and last four symbols are **pcodewords**. As it can be seen now codeword is $c_2 = [ep_1, ep_2, ep_3, c_{21}, c_{22}, c_{23}, c_{24}, c_{25}, c_{26}]$

The section presented a particular technique of header introduction and gave intuition on why the proto messages' approach provides overlap. The next section illustrates how exactly the algorithm perceives the data containing valid reads. Later sections will empirically prove this in terms of a toy example.

5.3. Assembling DNA sequence using valid codewords

In order to provide enough overlap between codewords, we introduced the concept of proto messages. Why does the concept of proto messages work in the task of assembling DNA sequence? Let us answer the question using a concrete setup and example described below. Everything that has been introduced until this section:

- As it was described in Chapter 3, we introduced the concept of **error control codes** layer between biological and computational layers in DNA sequencing. We enforced the length of reads to be the same as codewords - which is why we called them **n-mers** and tried the conventional approach of transferring messages.
- Codewords that have been transferred without error were recognized as valid. Since the overlaps are needed for sequencing algorithms to work, codewords alone do not fit as data.

- In the previous section, the concepts of header and proto message were introduced.
- The previous section also described a useful property of cyclic codes that, in turn, led to the concept of proto messages.

Now that we have the setup described above with proto messages and the ECC layer in sequencing, let us discuss how they work. The updated schema is shown in Figure 5.6.

- Proto messages ($k = 7$) are encoded into sequence of $n = 9$ codewords, with length of header being $s = 3$. Codewords are concatenated together to form a DNA sequence: `<c1c2c3c4>` and passed to the synthesizer in order to get molecules for storage.
- As it is seen in Figure 5.7 two codewords are concatenated together:

`c2c3=ep1ep2ep3c21c22c23c24c25c26ep1ep2ep3c31c32c33c34c35c36,`

and are part of the DNA sequence.

- On the sequencer's **biological** layer reads are sequenced individually at a length $n = 9$. In this case, Figure 5.7 displays all the toy parametered example reads, that have been sequenced; as it can be seen 3 of them have errors, specifically $r5, r7$ and $r8$ (according to research paper [17] errors are normally closer to the end of sequence).
 - On the sequencer's **ECC** layer, erroneous reads are going to be detected using parity check matrix and discarded and valid reads are going to be preserved.
 - We take into account that guaranteed minimum overlap is going to be equal to 3 (since the header's length is $s = 3$), and all the other reads are discarded from the beginning. As it can be seen in Figure 5.8 by giving up certain overlap, we get ideal (valid) reads.
- In the previous chapter, we also introduced algorithms for sequencing that are based on overlapping parts of reads. Since the overlap between reads is **enough** for assembly, it is possible to utilize the de Bruijn graph or shortest common superstring to assemble a DNA sequence from codewords. Reads that are recognized as valid codewords are shown in Figure 5.8.
 - SCS is the algorithm of choice since our toy example only consists of $9 * 4 = 36$ symbols, and for de Bruijn assembly, it is recommended to use the overlap of $n - 1$.
 - s overlaps between reads/codewords are enough to reconstruct the sequence. Utilizing them algorithm reconstructs DNA sequence from the reads, and assembled sequence is later transferred to **decoder**.

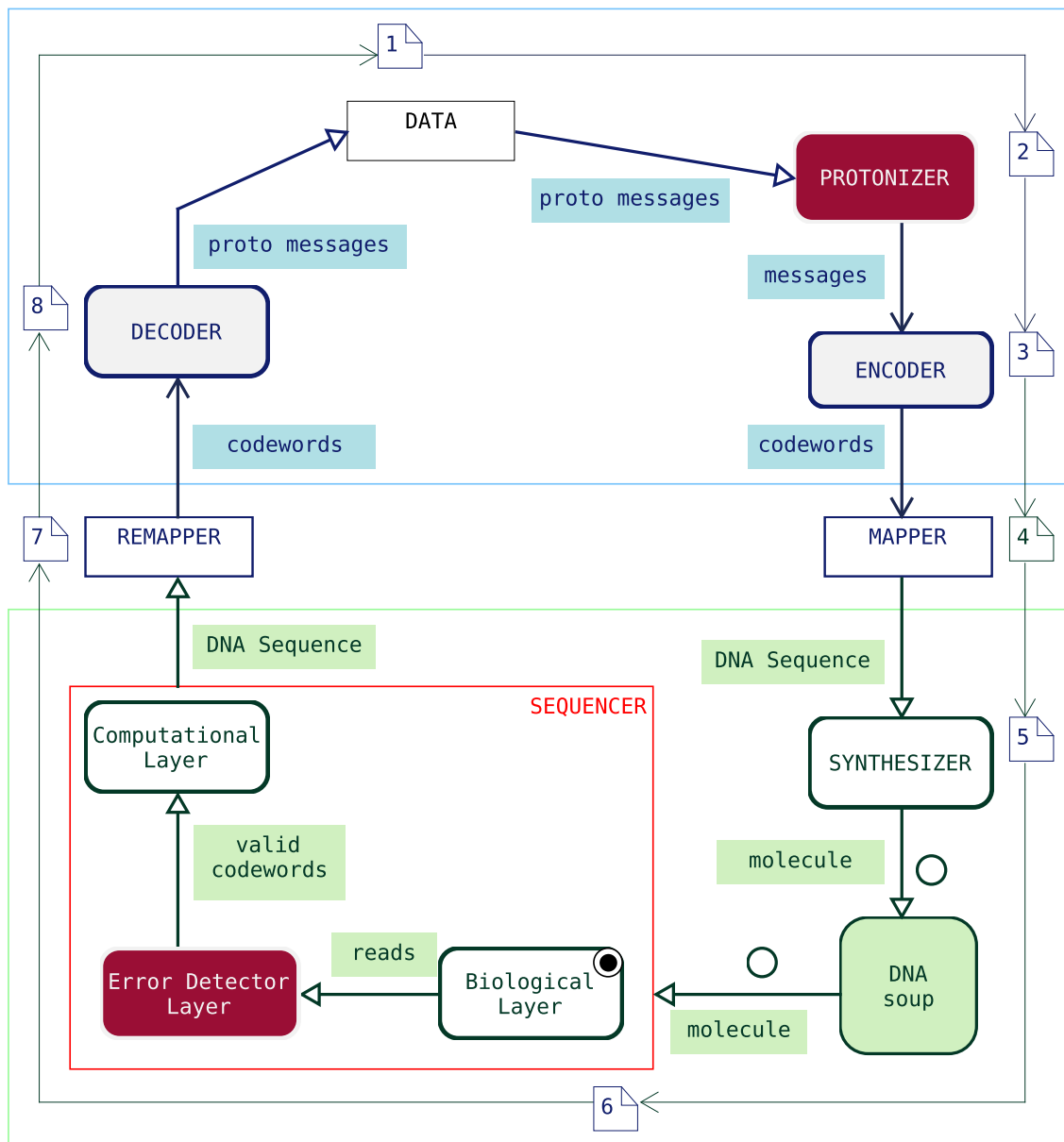


Figure 5.6.: Depiction of updated schema of DNA as a storage. As it can be seen on the figure new stages have been added to our initial schema of DNA as a storage. Here we have **PROTONIZER** and **Error Detector Layer** in sequencing.

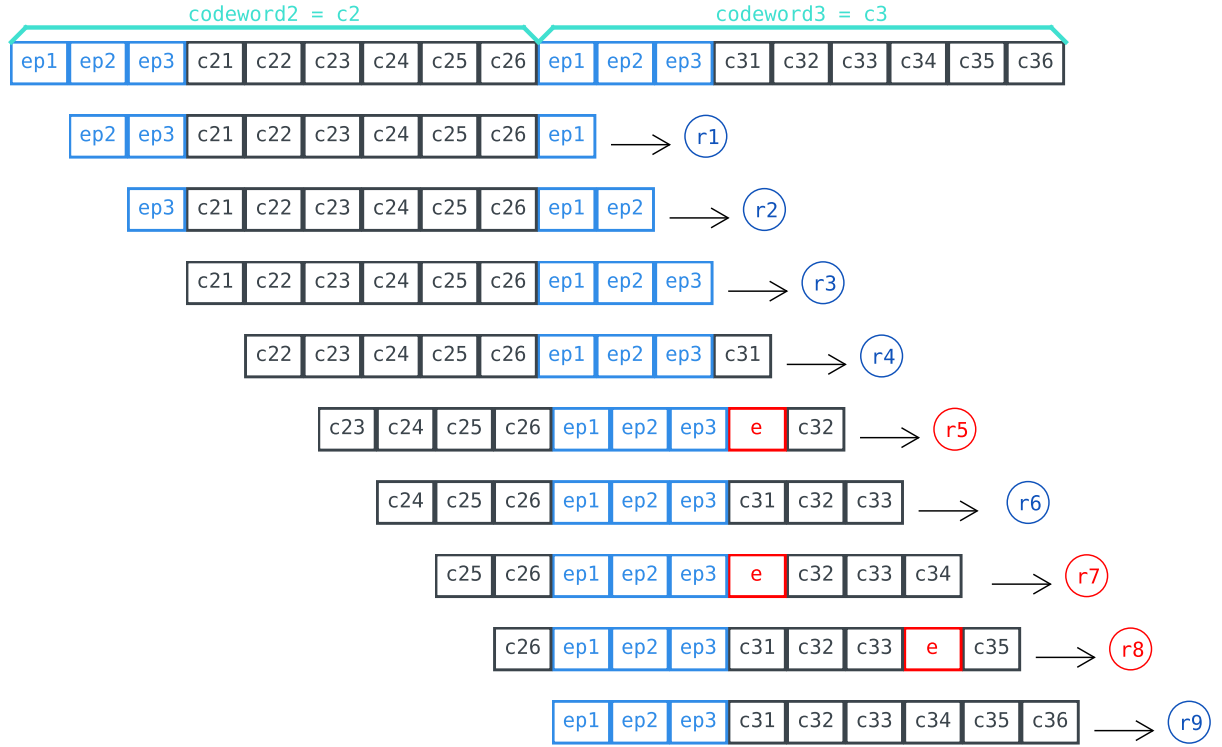


Figure 5.7.: Depiction of individual reads after sequencing unknown DNA sequence. Please note that the order of the reads is not defined; they are pretty much scattered. For better demonstration order of the reads is preserved. It is also seen that out of nine reads, three are erroneous, and six are valid.

As has been described in this section, we concentrated mainly on why the concept of proto messages works. The next section will experimentally make a comparison between the state-of-the-art/no coding implementation of DNA storage and the header/proto message approach.

Regardless of changes in simulation required to implement, old components remain in the system; instead, it is necessary to add new components like **Protonizer**. Also, notice that we are working on a "toy parameterized" example, and the length of messages is $k = 11$, length codewords $n = 15$. Additionally, it is required to perform a certain number of repetitive experiments. So, random data generation has also been automated in order to achieve such a result. The next section will describe the updated structure of source code, initial setup, and experiments themselves. Conclusively, an average of 3000 experiments will be taken, and the application will produce comparative analysis with plots.

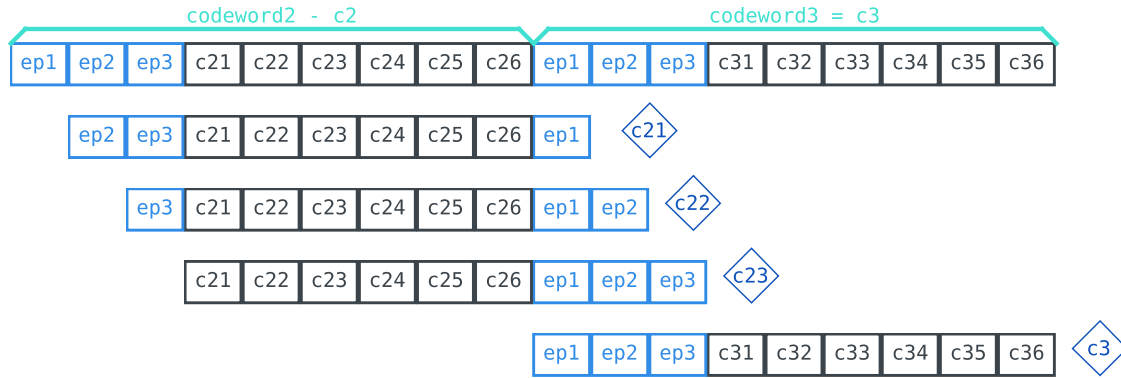


Figure 5.8.: Depiction of valid codewords preserved during sequencing. We discarded certain reads, however with a guarantee of assembly from **valid** reads, and discarding **erroneous** ones.

5.4. Empirical analysis on toy example

This section analyzes empirically the concepts discussed in previous sections by bringing a concrete working example of assembly from codewords on a simulated system. First, modifications to the original setup are going to be discussed. Then experiments are performed by simulating *substitutions* from sequencing errors and employing a particular randomized message generator. Conclusively, comparative graphs of both approaches are being generated.

5.4.1. Updated setup

As it was described in a prior section, a particular randomized message generator is going to be included in a system. The idea behind a randomized message generator is to produce a certain number of messages with defined length and automate the process. Figure 5.9 illustrates a definition of the message generator.

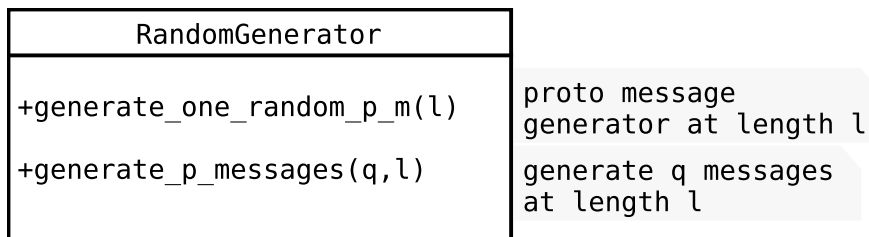


Figure 5.9.: Random proto message generator

In addition to the randomized proto messages generator, it is required to add one more component - Protonizer. Protonizer appends a unique pre-defined sequence to the beginning of proto messages, which in turn results into `<[static_head|proto_message]>`. Figure 5.10 displays the definition of Protonizer class.

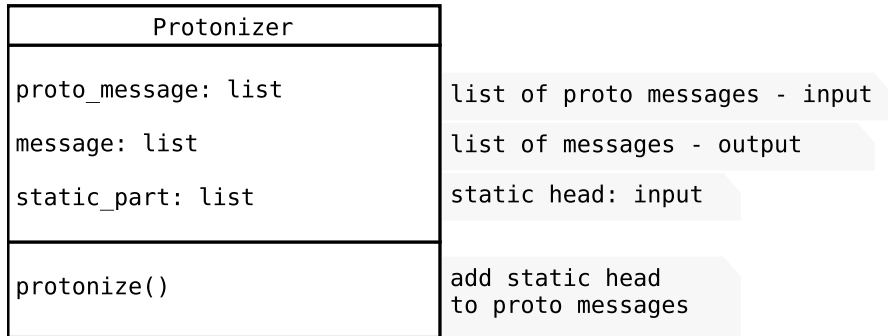


Figure 5.10.: Protonizer component - outputs messages given proto messages

Finally, it is essential to implement the channel that simulates errors introduced to the system by Sequencer at reads generation step. In order to be able to simulate such a system, it is required to revisit the error types introduced at the sequencing step. Errors introduced during DNA sequencing depend on the technology employed.

This thesis mostly took the approach from Illumina's single-read sequencing technology [30]; instructions, databases, and algorithms are available from open-source manuals on Illumina's website. Algorithms used are de Bruijn assembly and shortest common superstring. Errors introduced in sequencing are typically substitutions, and much less likely deletions and insertions. However, according to research papers, there are certain properties of the DNA sequence in the sequencing process, that increase error probability (mostly deletions and substitutions). Properties of the undefined DNA sequence are specifically the GC content and homopolymer stretches (same-symbol data, e.g., CCCCCC). In the case of this work, the examples that are tested are based on substitutions only. Algorithm for substitution-based error introduction is shown in Listing 5.1.

```
def distort_reads_strong_version(reads,p):

    #INPUT: reads - reads/codewords to be distorted
    #INPUT: p - probability given at single simulation
    #normally probability increases

    #iterate over all reads/codewords
    for i in length_Of_Reads:
```

```

#iterate over symbols in particular codeword
for j in single_codeword:

    #pick randomly value between 0 and 1
    s=pick randomly from [0 and 1]

    #if random value is less than probability
    #at single simulation
    if s<p:
        swap character to random
        from given to random in [A,C,T,T]

```

Listing 5.1: Channel algorithm: working principle is very similar to Digital Memoryless Channel

The above paragraphs, specified components that have been added to the updated system of DNA as storage. In the workflow manager, an additional method for the Protonizer element was added. Now that all components of the system are present, it is possible to proceed to the comparison of coding based DNA storage and state-of-the-art sequencing approach.

5.4.2. Simulations performed on toy parameterized example

Now that the system for experiments is all set, it can be collected together, and empirical analysis can be performed. In order to produce a comparative analysis with plots, additional `plt` library from *Matplotlib* is utilized.

Rules established for experiments are:

- Number of experiments is chosen to be 3000.
- At every experiment, a random set of proto messages is generated first for state-of-the-art (standard sequencing) approach and then the static header approach. Parameters are as follow:
 - **Lengths:** The length of a single proto message is 7; the length of a single message with static header is 11; the length of the codeword is equal to 15. Codewords and headers are not involved in experiments for standard sequencing.
 - **Quantity:** Quantity of proto messages per experiment is agreed to be 7.
 - **CRC Code Polynomial:** Polynomial for CRC code is $x^4 + x^3 + x^2 + 2x + 3$. Polynomial is not involved in experiments for standard sequencing.
 - **Header:** Header is chosen to be [1, 2, 1, 1, 2].
- At every experiment, error probability starts from 0 and raises to 1.0 with an interval of 0.05. Even though the likelihood of substitution errors is not as high as shown in setup, it has been decided to test all the cases on a toy parameterized example.

- All the parameters are assumed to be proportionate to state-of-the-art sequencing technologies and can be adjusted to a given real-world example.

Listing 5.2 shows the header/proto messages approach section of tests for updated workflow. Listing 5.3 shows the state of the art approach(it can be perceived as a continuation of proto messages part). As can be seen, the distinction between the listings is that in the conventional sequencing approach, there are five steps to perform, whereas the proto message/header has eight steps.

```
from workflow_manager import WorkflowManager
import matplotlib.pyplot as plt
from bioinformatics.work_without_ecc import *
from helper import *

#error probabilities from 0->1
ERROR_PROBABILITIES = [0,0.05,0.1,0.15,0.2,0.25,0.35,0.4,
0.45,0.5,0.55,0.60,0.65,0.70,0.75,0.8,0.9,1.0]

#number of experiments
N_E=3000

#sets figure size to 12x25
plt.figure(figsize=(12, 25))

#coefficients of CRC code
POLYNOM = [1,1,1,2,3]

#static part set to [1,2,1,1,2]
S_PART=[1,2,1,1,2]

#codeword length set to be 15
N=15

#iterate through number of experiments
for t in range(N_E):

    #generate proto messages: 7 per experiment
    proto_messages = generate_messages(7,7)

    #1) manager takes proto messages, polynomial and static part
    #2) joins them with static part
    #3) encodes messages
    #4.1) performs simulation of mapping(0,1,2,3->A,C,T,G)
    #4.2) performs virtual synthesis -> simple concatenation
    #5) passes through channel for substitutions
    #6) detects erroneous sequences and discards them
    #7) assembles using shortest common superstring
    #8) #errors_in_assembly/#of_total_bases

    wm = WorkflowManager(proto_messages,POLYNOM,S_PART,N)
    wm.generate_messages()
```



```

wm.encode()
wm.map_and_synthesize()

#iterate through error probabilities
for probability in ERROR_PROBABILITIES:

    #create distorted reads depending on error probability
    #at 0 there are no substitutions
    wm.create_distorted_reads(probability)

    #detect correct codewords
    wm.detect_codewords()

    #assemble sequence using shortest common superstring
    wm.assemble()

```

Listing 5.2: 3000 simulations on toy parametered example using coding theory with header/proto message approach.

```

from workflow_manager import WorkflowManager
import matplotlib.pyplot as plt
from bioinformatics.work_without_ecc import *
from helper import *

#error probabilities from 0->1
ERROR_PROBABILITIES = [0,0.05,0.1,0.15,0.2,0.25,0.35,0.4,
0.45,0.5,0.55,0.60,0.65,0.70,0.75,0.8,0.9,1.0]

#number of experiments
N_E=3000

#sets figure size to 12x25
plt.figure(figsize=(12, 25))

#iterate through number of experiments
for t in range(N_E):

    #generate proto messages: 7 per experiment
    proto_messages = generate_messages(7,7)

    #1) manager takes proto messages
    #2.1) performs simulation of mapping(0,1,2,3->A,C,T,G)
    #2.2) performs virtual synthesis -> simple concatenation
    #3) passes through channel for substitutions
    #4) assembles everything in the end
    #5) #errors_in_assembly/#of_total

    wmsoa = WorkflowManagerForSOA(proto_messages)
    wmsoa.map_and_synthesize()

```

```
#iterate through error probabilities
for probability in ERROR_PROBABILITIES:

    #create distorted reads depending on error probability
    #at 0 there are no substitutions
    wmsa.create_distorted_reads(probability)

    #assemble sequence using shortest common superstring
    wmsa.assemble()

    #proportion of errors to the initial length of DNA sequence
    e_b = wm.number_of_erroneous_bases()
```

Listing 5.3: 3000 simulations on toy parametered example without using coding theory and headers.

From the listings described above, Figure 5.11 is generated.

- Independent variable(x) is the probability of substitution errors - variates from 0 to 1 (maximum probability of failure).
- The dependent variable in the figure describes the proportion of quantity erroneous symbols to the length of DNA. Each point in the plot constitutes a cumulative average of 3000 experiments.
- As it is shown in the figure at some point, the approach proposed in this thesis for the particular case of 3000 experiments until 50% substitution error probability achieves a lower error rate in assembly. Starting from 53% performance of the static header approach worsens. Eventually, at 80%, graphs meet, and the conventional method has a better outcome.

5.5. Conclusion

As it can be seen in Figure 5.11, the approach introduced in this thesis offers specific improvements. However, it is necessary to take into account that the model presented here was produced on a toy parameterized example, where parameters are assumed to be proportional to the real-world application of sequencing. There are also certain facets in the biological layer in the sequencing step that were not taken into account in this thesis. The situation was assumed to be ideal, where substitution errors were tested.

It is necessary to take into account that the field the thesis dealt with is purely based on mathematical models. There are only three labs that implemented DNA as a storage system experimentally. Furthermore, the work can serve as a good starting point for implementing a real-world system.

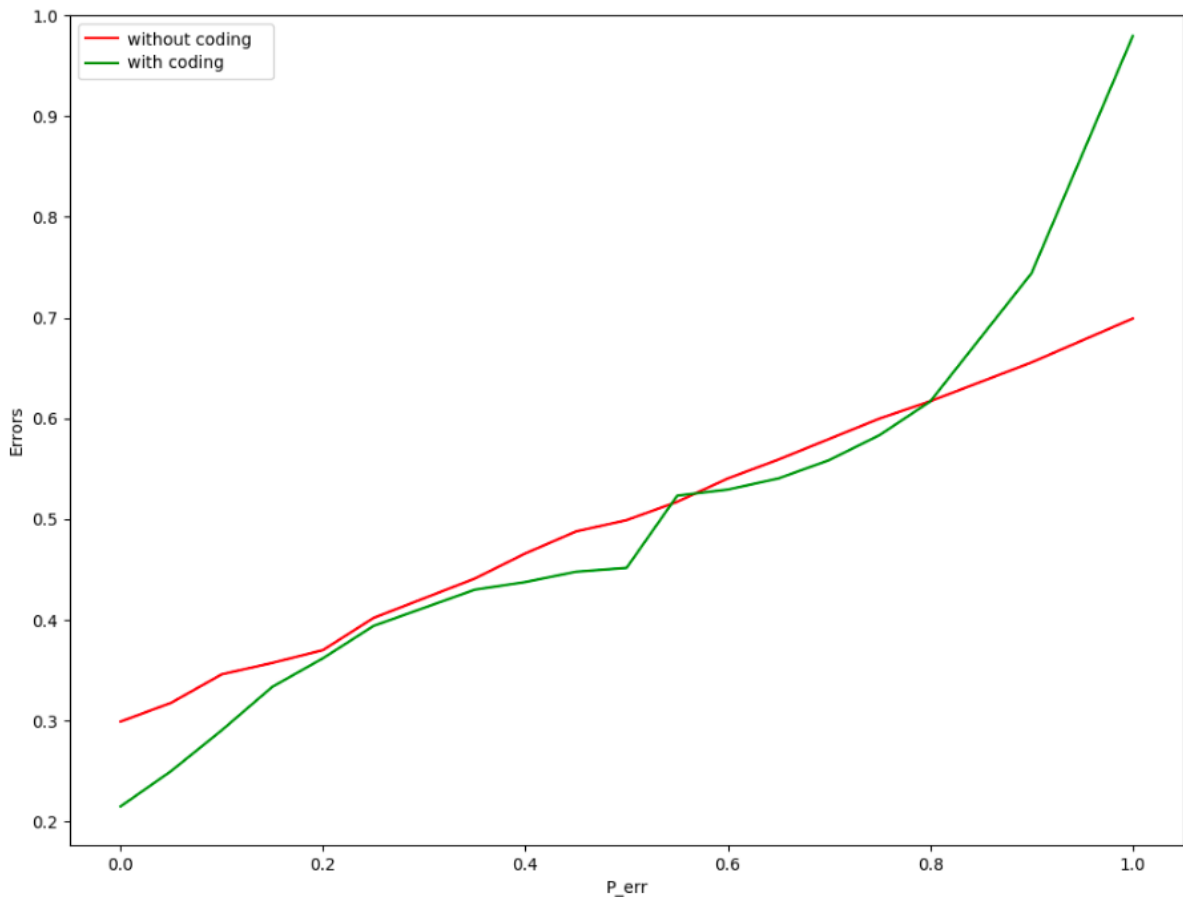


Figure 5.11.: Comparative plot of two different approaches to DNA as a storage. Red graph corresponds to conventional approach. Green plot corresponds to graph message/header approach.

6. Future outlook

In this thesis, we compiled literature about DNA as storage and covered the basics of sequencing algorithms and biology for electrical engineers. Additionally, we provided a particular technique for detecting erroneous reads and preserving overlaps for assembly of DNA sequence.

There are many various tests and experiments are left for the future. Even though a particular implementation of simulation is provided, real-world parameters and components are needed in order to conduct more in-depth experiments. Furthermore, there are many other concepts, such as quality of reads (coverage) and approaches for sequencing problems (e.g., overlap layout consensus and hybrid methods) that can be discussed and tested. Another topic to be discussed is the biochemical aspect of the project, such as synthesis and chemical contents of DNA soup.

Finally, the topic of DNA as storage can lead to the creation of a lab dedicated specifically to that subject.

A. Appendix

In chapter 3 we briefly introduced the way codes are organized. Appendix is about the main tools used in DNA as a storage machinery. In order to simulate system it was necessary to use certain programming tools and approaches. As it can be seen from our figures our DNA as a storage setup is based on components from two fields: error control codes and bioinformatics, so it is intuitive to utilise object oriented programming techniques in order to simulate the setup. As a programming language in this work we decided to utilise python for two reasons: ease of use and possibility to reuse algorithms for bioinformatics. This section will give a brief description of **structural viewpoint** or in other words class diagrams.

Use this (and any self-created) sections to insert the appendix.

A.1. Tools used for DNA as a storage

A.1.1. Error control codes package

For error control codes package we have two classes: **Encoder** and **Detector**(since our Channel is special, we put it in helpers package) as shown in figure A.1. Each one of those classes has corresponding methods that serve specific purpose; every method will be described in corresponding code listings.

- **Encoder** - contains three methods: `construct_generator_matrix()`, `encode` and `encode_messages`. Description of methods is shown in listing A.1.

```
class Encoder(object):
    """constructor in Python language:
        messages: collection of messages to be encoded
        polynom: CRC polynomial to generate G-matrix with
        k: length of each message
        n: length of each codeword
    """
    def __init__(self, messages, polynom, k, n):
        ...

    #Private helper method constructs generator matrix
    def __construct_generator_matrix():
        ...
        return generator_matrix
```

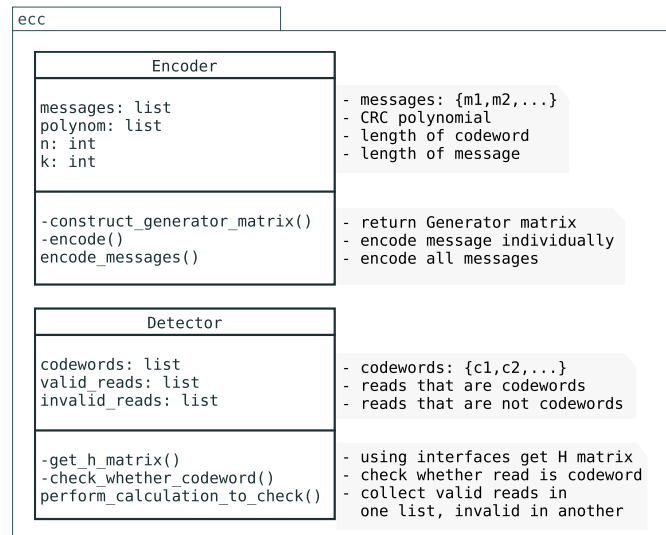


Figure A.1.: Error control codes package; contains two classes only: Encoder and Detector.

```

"""Private encode method
Returns: encodes single message out of collection
of messages and returns codeword """
def __encode():
    ...
    return codeword

"""Returns: using helper methods encodes all the messages;
uses pass-by-value approach"""
def encode_messages():
    ...

```

Listing A.1: Description of each method; implementation is omitted. Dark-green italic font represents comments in python.

- Detector - contains three methods: `check_whether_codeword()`, `get_h_matrix()` and `perform_calculation_to_checks`. Description of methods is shown in listing A.2:

```

class Detector():

    #codewords: array of codewords fed in our system
    def __init__(self, codewords):
        ...

    #returns h_matrix using cross-class interfaces
    def __get_h_matrix():
        ...
        return h_matrix

```

```
#checks whether each individual read is codeword  
#returns True or False  
def __check_whether_codeword(self):  
    ...  
    return status  
  
#method used for all codewords  
#collects valid codewords to  
def perform_calculation_to_check(self):  
    ...
```

Listing A.2: Description of each method from Detector class; implementation is omitted. Dark-green italic font represents comments in python.

As it is seen from implementations above package was designed with

A.1.2. Error control codes package

For error control codes package we have two classes: **Encoder** and **Detector**(since our Channel is special, we put it in helpers package) as shown in figure 4.2. Each one of those classes has corresponding methods that serve specific purpose; every method will be described in corresponding code listings.

Bibliography

- [1] D. Reinsel, J. Gantz, and J. Rydning, “The Digitization of the World From Edge to Core,” 2018. (p. 1)
- [2] Y. Dong, F. Sun, Z. Ping, Q. Ouyang, and L. Qian, “DNA storage: research landscape and future prospects,” *National Science Review*, vol. 7, pp. 1092–1107 vol.7, Jan. 2020. DOI: 10.1093/nsr/nwaa007 (p. 1)
- [3] S. M. H. T. Yazdi, H. M. Kiah, E. R. Garcia, J. Ma, H. Zhao, and O. Milenkovic, “DNA based storage: Trends and methods,” *CoRR*, vol. abs/1507.01611, 2015. (p. 1)
- [4] C. Takahashi, B. Nguyen, K. Strauss, and L. Ceze, “Demonstration of end-to-end automation of DNA data storage,” *Scientific Reports*, vol. 9, Dec. 2019. DOI: 10.1038/s41598-019-41228-8 (p. 1)
- [5] R. Grass, R. Heckel, M. Puddu, D. Paunescu, and W. Stark, “Robust chemical preservation of digital information on DNA in silica with error-correcting codes,” *Angewandte Chemie International Edition*, vol. 54, p. 2552, Jan. 2015. DOI: 10.1002/anie.201411378 (p. 1)
- [6] N. Seeman, “An overview of structural DNA nanotechnology,” *Molecular Biotechnology*, vol. 37, no. 3, pp. 246–257, Nov. 2007. DOI: 10.1007/s12033-007-0059-4 (p. 1)
- [7] A. Travers and G. Muskhelishvili, “DNA structure and function,” *The FEBS journal*, vol. 282, Apr. 2015. DOI: 10.1111/febs.13307 (p. 2)
- [8] H. Lodish, A. Berk, and S. Zipurski, *Molecular Cell Biology. 4th edition.* New York: Freeman, W.H., 2000. (p. 4)
- [9] R. Hughes and A. Ellington, “Synthetic DNA synthesis and assembly: Putting the synthetic in synthetic biology,” *Cold Spring Harbor Perspectives in Biology*, vol. 9, p. a023812, 01 2017. DOI: 10.1101/cshperspect.a023812 (p. 5)
- [10] L. França, E. Carrilho, and T. Kist, “A review of DNA sequencing techniques,” *Quarterly reviews of biophysics*, vol. 35, pp. 169–200, 06 2002. DOI: 10.1017/S0033583502003797 (p. 6)
- [11] V. Dominguez Del Angel, E. Hjerde, L. Sterck, S. Capella-Gutierrez, C. Notredame, O. Vinnere Pettersson, J. Amselem, L. Bouri, S. Bocs, C. Klopp, J. Gibrat, A. Vlasova, B. Leskosek, L. Soler, M. Binzer-Panchal, and H. Lantz, “Ten steps to get started in genome assembly and annotation,” *F1000Research*, vol. 7, no. 148, 2018. DOI: 10.12688/f1000research.13598.1 (p. 7)

- [12] A. Lenz, P. H. Siegel, A. Wachter-Zeh, and E. Yaakobi, “Coding over sets for DNA storage,” *CoRR*, vol. abs/1812.02936, 2018. [Online]. Available: <http://arxiv.org/abs/1812.02936> (p. 8)
- [13] V. Guruswami, A. Rudra1, and M. Sudan, *Essential Coding Theory*. National Science Foundation, 2018. (p. 9)
- [14] J. C. Freeman, *Introduction to Forward-Error-Correcting Coding*. National Aeronautics and Space Administration, 1996. (p. 9)
- [15] C. Senger, *Lecture notes in error control coding: Algebraic and convolutional codes*. Institute of Telecommunications, University of Stuttgart, 2018. (p. 10)
- [16] W. W. Peterson and D. T. Brown, “Cyclic codes for error detection,” *Proceedings of the IRE*, vol. 49, no. 1, pp. 228–235, 1961. (p. 11)
- [17] R. Heckel, G. Mikutis, and R. Grass, “A characterization of the DNA data storage channel,” *Scientific Reports*, vol. 9, Mar. 2018. DOI: 10.1038/s41598-019-45832-6 (p. 13), (p. 33)
- [18] E. Yaakobi, “Storage meets DNA: How to fit a data center into a shoebox,” Technion, Israel Institute of Technology, Tech. Rep., Feb. 2019. [Online]. Available: <https://marconisociety.org/storage-meets-dna-how-to-fit-a-data-center-into-a-shoebox/> (p. 14)
- [19] G. van Rossum, *Python webpage*, 2020. [Online]. Available: <https://www.python.org/> (p. 16)
- [20] T. Goldschmidt, S. Becker, and E. Burger, “Towards a tool-oriented taxonomy of view-based modelling,” *Lecture Notes in Informatics (LNI), Proceedings - Series of the Gesellschaft fur Informatik (GI)*, pp. 59–74, 01 2012. (p. 16)
- [21] Wikipedia contributors, “Unified modeling language — Wikipedia, the free encyclopedia,” 2020, [Online; accessed 1-July-2020]. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Unified_Modeling_Language&oldid=965164174 (p. 16)
- [22] Wikibooks, “Object oriented programming — wikibooks, the free textbook project,” 2020. [Online]. Available: https://en.wikibooks.org/w/index.php?title=Object_Oriented_Programming&oldid=3693242 (p. 17)
- [23] S. group Illumina, *FASTQ files explained*, 2020. (p. 20)
- [24] Pevzner, Pavel and Compeau, Phillip, *Bioinformatics Algorithms: An Active Learning Approach, How do we assemble genomes?*, 2020, [Free online book]. [Online]. Available: <https://www.bioinformaticsalgorithms.org/bioinformatics-chapter-3> (p. 21)
- [25] K.-J. Räihä and E. Ukkonen, “The shortest common supersequence problem over binary alphabet is np-complete,” *Theoretical Computer Science*, vol. 16, no. 2, pp. 187 – 198, 1981. DOI: [https://doi.org/10.1016/0304-3975\(81\)90075-X](https://doi.org/10.1016/0304-3975(81)90075-X) (p. 22)

- [26] Wikipedia contributors, “Hamiltonian path problem — Wikipedia, the free encyclopedia,” 2020, [Online; accessed 1-July-2020]. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Hamiltonian_path_problem&oldid=965303601 (p. 22)
- [27] D. Maier, “The complexity of some problems on subsequences and supersequences,” *J. ACM*, vol. 25, no. 2, p. 322–336, Apr. 1978. DOI: 10.1145/322063.322075 (p. 24)
- [28] P. Compeau, P. Pevzner, and G. Tesler, “How to apply de bruijn graphs to genome assembly,” *Nature biotechnology*, vol. 29, pp. 987–91, 11 2011. DOI: 10.1038/nbt.2023 (p. 24)
- [29] Wikipedia contributors, “Eulerian path — Wikipedia, the free encyclopedia,” 2020, [Online; accessed 1-July-2020]. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Eulerian_path&oldid=956915794 (p. 24)
- [30] Illumina Technologies, “Genomic sequencing,” 2010. [Online]. Available: https://www.illumina.com/documents/products/datasheets/datasheet_genomic_sequence.pdf (p. 37)

Erklärung

Hiermit erkläre ich, dass ich diese Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie alle wörtlich oder sinngemäß aus anderen Werken übernommenen Aussagen als solche gekennzeichnet habe.

Die Arbeit wurde bisher keiner anderen Prüfungsbehörde vorgelegt und auch noch nicht veröffentlicht. Das elektronische Exemplar stimmt mit den gedruckten Exemplaren überein.

Stuttgart, 03. Juli 2020

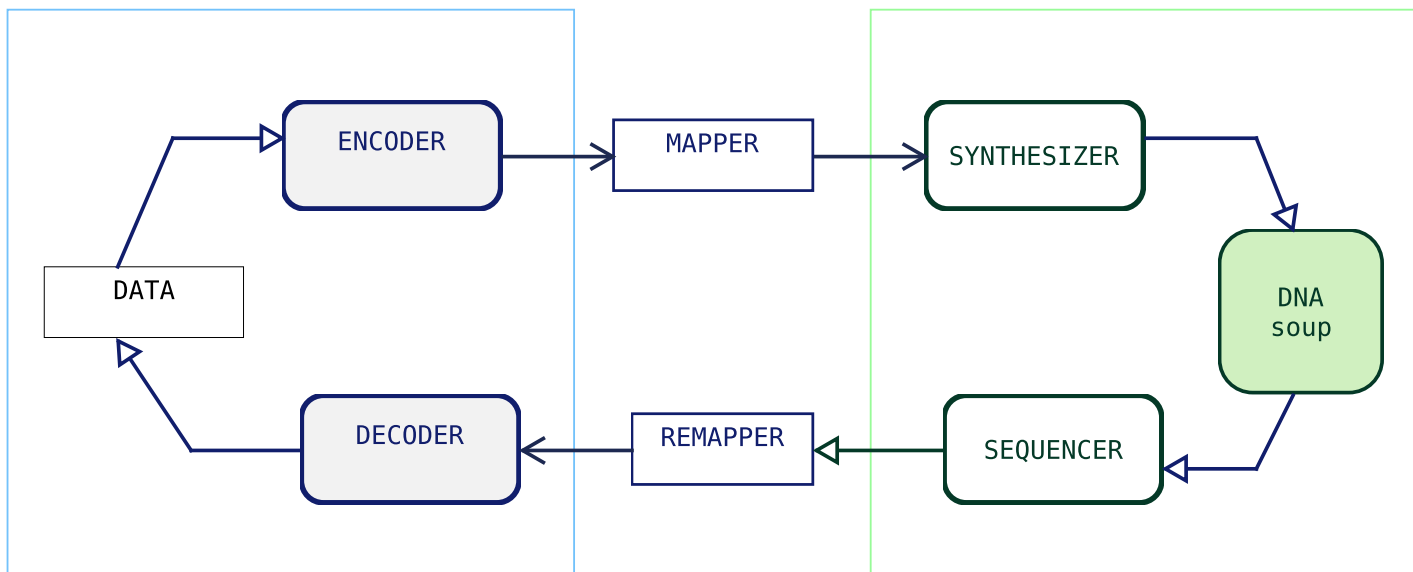
Fikrat Talibli



Master Thesis

DNA as a storage - Error Detection based on Cyclic Codes

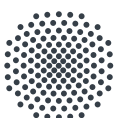
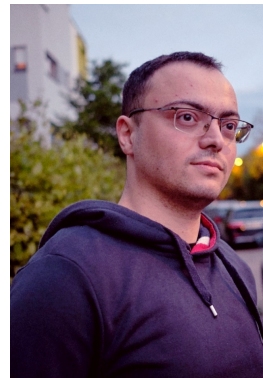
Fikrat Talibli



Date of hand in: July 13, 2020
Supervisor: Dr -Ing. Christian Senger

Abstract

With or without author's photograph. Here you have to add a (shortened) version of the abstract in the language of your thesis (English or German). You have to take care that this text is not too long and stays on one page in the output PDF. The area to the right of this text is reserved for your photograph. If you don't want to add one, please make the corresponding setting to utilize the whole available page width.



University of Stuttgart
Institute of Telecommunications